



<http://researchspace.auckland.ac.nz>

ResearchSpace@Auckland

Copyright Statement

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

This thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of this thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from their thesis.

To request permissions please use the Feedback form on our webpage.

<http://researchspace.auckland.ac.nz/feedback>

General copyright and disclaimer

In addition to the above conditions, authors give their consent for the digital copy of their work to be used subject to the conditions specified on the Library Thesis Consent Form.

*Department of Electrical & Computer Engineering
Software Engineering
The University of Auckland
New Zealand*

A Flexible Software Process Model

Diana Kirk

April 2007

Supervisor: Associate Professor Ewan Tempero



A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE RE-
QUIREMENTS OF DOCTOR OF PHILOSOPHY IN ENGINEERING

The University of Auckland

Thesis Consent Form

This thesis may be consulted for the purpose of research or private study provided that due acknowledgement is made where appropriate and that the author's permission is obtained before any material from the thesis is published.

I agree that the University of Auckland Library may make a copy of this thesis for supply to the collection of another prescribed library on request from that Library; and This thesis may not be photocopied other than to supply a copy for the collection of another prescribed library.

Signed:

Date:

Abstract

Many different kinds of process are used to develop software intensive products, but there is little agreement as to which processes give the best results under which circumstances. Practitioners and researchers believe that project outcomes would be improved if the development process was constructed according to project-specific factors. In order to achieve this goal, greater understanding of the factors that most affect outcomes is needed. To improve understanding, researchers build models of the process and carry out studies based on these models. However, current models contain many ambiguities and assumptions, and so it is not clear what the results of the studies mean. The statement of this thesis is that it is possible to create an abstraction of the software development process that will provide a mechanism for comparing software processes and software process models. The long term goal of the research is to provide planners with a means of tailoring the development process on a project by project basis, with the aim of reducing risk and improving outcomes.

Contents

1	Introduction	1
1.1	Research Area Overview	1
1.2	Problem to be Addressed	2
1.3	Modelling for Understanding	4
1.4	Thesis Statement	5
1.5	Approach	6
1.6	Overview of Contributions	7
1.7	Terminology	8
1.8	Document Roadmap	8
2	Software Development Processes	11
2.1	Nature of Software Development	11
2.2	Processes Overview	14
2.2.1	Waterfall	14
2.2.2	Spiral	15
2.2.3	Rational Unified Process (RUP)	16
2.2.4	Cleanroom	17
2.2.5	Agile alliance	19
2.2.6	XP	19
2.2.7	Scrum	20
2.2.8	Crystal	21
2.2.9	Feature Driven Development (FDD)	21
2.3	Categorising Processes	21
3	Software Process Models	25
3.1	Predictive Modelling	26
3.1.1	Cost estimation	26
3.1.2	Fault prediction	27

3.1.3	Limitations	28
3.2	Controlled Experiments	30
3.2.1	Limitations	31
3.3	Simulation Modelling	31
3.3.1	System dynamics	33
3.3.2	Event driven	34
3.3.3	State based	35
3.3.4	Limitations	35
3.4	Discussion	37
3.4.1	Software measurement	37
4	Related Work	39
4.1	Process Frameworks	40
4.1.1	The Spiral model	40
4.1.2	The OPEN process	40
4.1.3	The Rational Unified Process (RUP)	42
4.2	Process Tailoring Approaches	42
4.2.1	Basili and Rombach: Tailoring to Project Goals and Environments	42
4.2.2	Boehm and Turner: Balancing Agility and Discipline	44
4.3	Process Simulation	44
4.3.1	Drappa and Ludewig: Interactive simulation	45
4.3.2	Lahey: Project management	46
4.3.3	Munch: Process patterns	46
4.3.4	Storrie: Process patterns	47
4.4	Experimental Frameworks	47
4.4.1	Kitchenham et. al.: preliminary guidelines	48
4.4.2	Basili et. al.: families of experiments	49
4.4.3	Williams et. al.: XP-EF	50
4.5	People Factors	50
4.5.1	Layered behavioural model	51
4.5.2	Human competencies model	53
4.5.3	Team behaviour model	53
5	Case for a Theoretical Model	55
5.1	Research Perspectives	55
5.1.1	Modelling basics	55
5.1.2	Research basics	57

5.2	Case for a Theoretical Model	59
5.2.1	Need for a model	59
5.2.2	Support for a model	59
5.3	Approach	61
5.3.1	Objectives	61
5.3.2	Scope	63
5.4	Evidence Strategy	65
6	Model Properties	67
6.1	Properties Source	67
6.1.1	Process characteristics	67
6.1.2	Model limitations	69
6.1.3	Real-world situations	69
6.2	Properties	70
7	A Model	73
7.1	Overview	73
7.2	KiTe Model	79
7.2.1	Project	80
7.2.2	Atomic types	81
7.2.3	Product	81
7.2.4	CapabilitySpec	84
7.2.5	Partition	85
7.2.6	Perspective	87
7.2.7	GoalsBenchmark	88
7.2.8	Engineer	89
7.2.9	Context	91
7.2.10	Method	92
7.2.11	Technique	94
7.2.12	ContextModel	96
7.2.13	Activity	98
7.2.14	RealisedProcess	99
8	Evidence	105
8.1	Evidence map	106
8.2	Capture all Processes and Process Models	107
8.2.1	Study 1: Waterfall process	111
8.2.2	Study 2: Coding variations	117

8.2.3	Study 3: XP process	120
8.2.4	Study 4: Collaborative programming field study	129
8.2.5	Study 5: Event-driven simulation model	133
8.2.6	Study 6: Pair programming classroom study	138
8.2.7	Study 7: State-based simulation model	140
8.2.8	Study 8: System dynamics simulation model	149
8.2.9	Study 9: Concurrent programming field study	160
8.2.10	Study 10: Variations in XP process	163
8.2.11	Study A-H: Miscellaneous process elements	166
8.3	Compare Processes and Process Models	168
8.3.1	Study 11: Developer collaboration	168
8.4	Discussion	171
9	Identifying Process Risks	173
9.1	Overview of Risk Management	173
9.2	Risks in XP Process	178
9.2.1	Single iteration	179
9.2.2	Process	182
9.2.3	Discussion	184
10	Evaluation	185
10.1	Evaluation against Criteria	185
10.2	Evaluation against Related Work	189
10.2.1	Process frameworks	189
10.2.2	Process tailoring	190
10.2.3	Process modelling	192
10.2.4	Experimental frameworks	193
10.2.5	People factors	194
10.3	Evaluation against Research Objectives	194
10.4	Evaluation of Approach	196
10.5	Discussion	197
11	Conclusions	201
11.1	Summary	201
11.2	Contributions	202
11.3	Future Work	203
11.3.1	Model foundation	203
11.3.2	Process representation	204

11.3.3	Process risk	204
11.3.4	Assumptions	205
11.3.5	Evidence	205
11.3.6	Product model	206
11.3.7	Process customisation	206
11.3.8	Predictive tool	206
11.4	And Finally	206
A	Glossary	209

List of Figures

5.1	Model objectives	62
5.2	Objectives for ‘Represent’	63
6.1	Product and process	70
7.1	Schematic overview of KiTe	74
7.2	KiTe Product	75
7.3	KiTe Method and Technique	77
7.4	KiTe model relationships	80
7.5	State machine for KiTe	101
7.6	Disturbing a Method	102
8.1	KiTe goal hierarchy	108
8.2	Goal hierarchy for ‘Capture’	109
8.3	Goal hierarchy for ‘Miscellaneous’	110
8.4	Simple waterfall	114
8.5	Waterfall with defect resolution	115
8.6	Coding in a waterfall process	118
8.7	Coding before designs completed	118
8.8	Coding and fixing designs	119
8.9	Coding in an XP process	119
8.10	XP process iteration	124
8.11	Collaboration results in KiTe	132
8.12	PP and TDD simulation results in KiTe	137
8.13	PP and TDD Technique relative performance	137
8.14	State-based simulation baseline	144
8.15	Concurrency Technique relative performance	162
8.16	XP process iteration	164

8.17 XP - Engineers with low skill levels	165
8.18 Case study overview	169

List of Tables

7.1	State transition diagram for RealisedProcess	103
8.1	Waterfall Product Model	112
8.2	Waterfall Context Model	113
8.3	XP Activities	121
8.4	XP Product Model	122
8.5	XP Engineer and Context Models	123
8.6	XP Practices in KiTe	128
8.7	Collaboration results (Nosek)	130
8.8	Collaboration Product Model	130
8.9	Collaboration CodeRequirements Method	131
8.10	Collaboration Engineer and Context Model	132
8.11	PP and TDD adoption results (Melis et. al.)	134
8.12	PP and TDD Product Model	135
8.13	PP and TDD DesignAndCodeAll Method	136
8.14	State-based simulation Product Model (Raffo et. al.)	142
8.15	State-based As-Is calculations	146
8.16	State-based To-Be calculations	147
8.17	State-based Activity effectiveness summary	148
8.18	System dynamics Baseline - Incr A Activities	155
8.19	System dynamics Baseline - Incr B Activities	155
8.20	System dynamics Baseline - Incr C Activities	156
8.21	System dynamics Case n1 - Increment A Activities	156
8.22	System dynamics Case n3 - Increment A Activities	157
8.23	System dynamics Case n5 - Increment A Activities	158
8.24	System dynamics Case Optimal - Increment A Activities	158
8.25	System dynamics Case n6 - Increment A Activities	159
8.26	Productivity v. concurrency level (Parrish et. al.)	160

8.27 Concurrency Product Model 161
8.28 Concurrency Engineer and Context Model 163

Acknowledgements

I would like to thank Ewan Tempero, my supervisor for this dissertation, for his superb support and guidance during the time of my research. Ewan always succeeded in providing advice and direction that were appropriate and timely and was always available for discussion. I am very grateful for his encouragement.

During the time of this research, I attended a number of conferences and workshops to present papers based on the research. I would like to thank Ita Richardson, Juan Ramil, Mary Shaw, James Noble and Stephen MacDonell for taking the time to provide me with feedback on some of these papers and Judith Segal, David Weiss and Colin Coghill for some interesting and, from my perspective, fruitful discussions about various aspects of the research. I would also like to thank Barbara Kitchenham for her willingness to share with me some of her vast knowledge relating to research techniques and evidence and thus helping me on my beginner's journey towards an understanding of the current state of software research.

Finally, I would like to thank the Department of Computer Science at the University of Auckland for providing me with funding to support my research and the department staff for providing an excellent atmosphere to work in.

1

Introduction

1.1 Research Area Overview

Many researchers and practitioners are interested in exploring different ways of producing software-intensive products. The reason is that it is generally agreed in the software industry that the kind of process used in a software project is a key factor in determining what are the outcomes for the project. Example outcomes are the ability of the project to deliver the software product on-time and within budget.

At the present time, there are a number of different kinds of process in use in industry. These are often categorised as either ‘traditional’ (commonly referred to as ‘heavyweight’) or ‘lightweight’. Traditional processes were created to help control very large software projects spanning several years, many of which exhibited safety-critical or other ‘large loss’ aspects. These processes are based on a manufacturing paradigm and are characterised by a phased approach, in which, for example, design tasks are strictly separated from coding tasks. Different phases tend to be carried out by different people, for example, ‘systems analysts’, ‘architects’, ‘coders’ and ‘testers’. As a result of the strict separation, large quantities of documentation are required to communicate decisions among the various parties. The well-known ‘Waterfall’ model represents an example of this kind of process.

The ‘light’ processes have emerged more recently as a response to the perceived ineffec-

tiveness of the traditional methods when applied to, for example, Web development. These processes tend to be more responsive to change in product requirements and are characterised by a strong people focus. Because of the close relationships between developers and customers, the underlying development paradigm for these methodologies is presented as ‘software-as-a-service’ and communications are generally face-to-face. ‘XP’ (eXtreme Programming) is an example of this kind of process.

Traditional and light processes are most commonly applied to different kinds of projects. Traditional processes are most often used in very large projects or for safety-critical products. Light processes are most often used for small projects or products that can be produced and changed quickly, for example, Web applications. However, there is much discussion about how to apply individual processes to projects other than those for which they appear to be most suited. For example, is it appropriate to use an XP process for developing a product that is expected to undergo further change in the future i.e. is part of a product line? There is also much discussion about the possibility of embedding elements of one process into another. For example, would deadlines be more likely to be met if a ‘Pair Programming’ technique from XP was used within a Waterfall process?

In this dissertation, I address the possibility of synthesising a new process from elements of existing processes. In order to achieve this, I study the work of those researchers who model the software process for the purpose of predicting outcomes and uncover limitations of the models that render the models inappropriate as a basis for synthesis. I present an abstraction of the process that supports such synthesis.

1.2 Problem to be Addressed

It is widely acknowledged in the software industry that no one process is appropriate for all software development projects [10, 35, 38, 57]. Some believe that each kind of process is appropriate for specific kinds of project and should be used only within such projects. This requires an assumption that any project can be classified as one of a number of discrete types, each with fixed boundaries. Others are adamant that their ‘favourite’ process may be applied to any project, with only minor adaptations required. Of greater interest is the possibility of synthesising new processes from existing ones. Many researchers and practitioners believe that the chance of project success can be improved by selecting process elements from different processes in order to tailor the process according to project-specific factors [13, 35, 46, 93, 115, 144, 153].

The interest in customisation comes from two directions. The first is the ‘traditional’ versus ‘light’ discussions [16, 21]. Practitioners understand that different kinds of process have

different strengths, but would like to know under which circumstances elements from one process may be embedded in another. The kinds of questions asked include: “How well does XP perform if the customer is unable to be on-site?”; “How would delivery schedules be affected if developers practice Test Driven Design within a Waterfall process?”. The second source of interest is from the study of software economics. The suggestion here is that a project should maximise value creation [102] and should use a process that is no more costly than necessary [24]. The kinds of questions that represent this kind of interest include: “What will be the effect on the quality of the delivered product if we replace code inspections by automated checking to reduce cost?”; “Can we customise a process to give best outcomes for a specific project by combining elements from existing processes?”.

Although the industry would like to answer the kinds of questions illustrated above, this is not possible at the present time. Before we can answer such questions, we must first be able to represent any process element from any process in a way that facilitates composition and prediction. It is contended here that no suitable abstraction exists.

One reason is that the problem space is not yet well-understood. Current processes have emerged in response to perceived need and, although attempts have been made to understand what are the key factors that affect outcomes, there is little data to support claims. Efforts have been made to collect supporting data, but the complexity of the problem space has rendered this difficult. In addition to the many technical challenges presented by a fast-changing industry, software process tasks are carried out by people rather than machines, and so issues of psychology and social behaviour are relevant.

It is now generally accepted that human factors, for example, management style and developer experience and motivation, have a major impact on the success of a software development effort [3, 24, 37, 34, 155, 157]. Curtis et. al. suggested in 1988 that process problems were overwhelmingly caused by people-related factors and recognised at that time the need for a behavioural model of the software development process [36]. However, there are no such models on which to base formal research into the effects of human factors on outcomes and current processes either assume a tendency to the mean or make assumptions about which factors are key. For example, the Waterfall process does not include any consideration of human factors. This is perhaps because of its traditional use for very large projects where human effects ‘average out’ over the course of the project. XP, on the other hand, embeds assumptions about developer performance, for example, that all developers work more efficiently and effectively together than alone. Although this represents some consideration of human factors, there is no mechanism for accommodating differences between developers.

In an attempt to accumulate data to increase the industry’s ability to predict process outcomes, researchers have carried out different kinds of studies. There are a number of concerns that relate to these studies. One such concern is the issue of how to measure the various factors

and attributes that apply to the software process. In 1995, Kitchenham, Pfleeger and Fenton identified a lack of integrity in the way in which software practitioners and researchers measure software-related attributes [89]. It is important to work with validated software metrics and at that time there was no agreed way to perform such validation. The authors made a plea for the industry to agree on a way to bridge the gap between measurement theory and software metrics. Although the plea appeared to spawn some heated discussions, it seems that no consensus has been reached, and the industry continues to work with metrics based on disputed foundations [91]. A second concern is that, as software engineering is a relatively immature discipline, researchers have not yet learned to routinely apply sound practices when conducting studies, and so resulting data is scarce, fragmented and of varying quality [7, 14, 46, 90, 125]. Gilmore describes four modes of research data collection and states that only one of these, hypothesis testing, results in establishing causal connections. He and others agree that, in order to carry out this kind of research, a theoretical framework is essential from which to spawn hypotheses [38, 55, 90]. As discussed above, there is no suitable abstraction for elements of a software process and so it is difficult to carry out hypothesis testing experiments and establish causal connections.

In summary, the industry at the present time has no abstraction or theoretical framework for software development processes. This means that practitioners are unable to combine process elements and predict outcomes and researchers find it difficult to investigate causal relationships.

1.3 Modelling for Understanding

In the previous Section, I described a problem of lack of a theoretical model of the software process. In this Section, I consider the act of model building from an historical perspective and conclude that an appropriate model should be explanatory rather than predictive and that such a model will be holistic rather than fragmented.

Rivett [139], when describing the status of model building in the field of operations research in 1972, reminds us that, throughout history, man has constantly searched for pattern and generalisation. From around 700 BC, the Babylonians measured and recorded the motions of the stars and planets, analysed these, and were successful in forecasting planetary events with great precision. Their recordings of hundreds of years of planetary data enabled them to estimate the value of the motion of the sun from the node with an error of only five seconds. The large quantity of data collected by the Babylonians supported accurate prediction. In fact, two thousand years later, the same estimation, based on models of planetary motion, yielded an accuracy of only seven seconds. The observation is that large quantities of accurate data often yield more

accurate predictions than those based on models.

Although the Babylonians recorded events with care, they made no attempt to theorise. The Greeks, on the other hand, followed a different approach, and built first mechanical and then geometrical models of planetary motion in an attempt to understand and explain. However, their models were made up of a number of parts and the Greeks had no success in unifying these. When applied to the Babylonian data, the models were found to be incorrect [139]. This illustrates that it is often difficult to achieve consistent results when a fragmented approach is taken i.e. a model of a part of a system may yield results that are invalid in the context of the bigger system.

Rivett presents another example from more recent times that illustrates that consistent and complete results will be achieved only if a model is based on an underlying theory. Kepler proposed three laws of planetary motion based on data that had been collected by Tycho Brahe. He applied an elliptical model to the motion of the planets and from this model produced laws that appeared to work. No-one knew the fundamental reason why the laws worked. I notice that, as the laws were based on planetary data, these laws could not predict the movements of other celestial objects, for example, comets. Newton later brought some understanding to bear on celestial motion when he postulated a force that acted between all objects with mass in the universe. From this understanding and unification of ideas from physics and astronomy, he was able to show that orbits for celestial objects, for example comets, were not only elliptical, but could be hyperbolic and parabolic. He was able to predict accurately for all celestial bodies, show that Kepler's Laws were a special case of Newton's Laws and improve the accuracy of Kepler's calculations.

Rivett summarises by stating that a model may be predictive without being explanatory, but an holistic, explanatory model is always predictive. When I apply this idea to the software process, it follows that previous process data may be successfully used to predict the outcomes of future projects that are based on similar processes. I understand that, if we wish to predict in a more general way, our predictive models must be holistic and explanatory. This means our models must be able to represent any element of any process, including both existing elements and those defined at some future time.

I have identified the need to represent different process elements for synthesis and prediction. In order to meet these goals, I want to capture process elements in a descriptive way i.e. capture elements as they actually happen.

1.4 Thesis Statement

Before we can synthesise processes from existing and future process elements, we must first be able to capture elements of processes and process models. Before we can predict outcomes of applying process elements, we must first be able to compare the effects on these outcomes of different elements.

I believe that it is possible to capture software development processes and process models in a way that allows us to compare processes and process models for the purpose of constructing new processes.

My thesis is realised as a conceptual modelling framework, *KiTē*, the elements of which are themselves models. The framework supports capture of, and facilitates comparison and composition of, processes and process models.

1.5 Approach

I have postulated the need for a model of the software development process that allows capture of any element from any process or process model and facilitates comparison between, and composition of, elements. The long term goal for such a model is the ability to predict outcomes when process elements are combined in various ways. Rivett and others argue that a model for prediction must be holistic and explanatory. ‘Holistic’ suggests that all relevant aspects of the process, for example, behavioural aspects, must be included. ‘Explanatory’ suggests that the model should be based on a theoretical abstraction rather than on specific data.

The creation of such a model is difficult. If the industry is to create such a model, it must first identify what are the characteristics of existing processes that must be included in a representation and understand what are the limitations of existing predictive models that render them inappropriate for general process representation.

I thus examine the characteristics of existing processes and process models and create from this examination a set of ‘desirable properties’. These properties will act as preliminary criteria against which to judge any candidate model. This provides an informal mechanism for evaluation, in that the criteria are subjective in nature. The aim is to gain some confidence prior to any evidence-gathering attempt that the candidate model is likely to support the objectives of capture and comparison.

Once a candidate model is proposed and evaluated against the preliminary criteria, some evidence must be presented to support the thesis that the model supports capture and comparison for the purpose of synthesis and prediction. As there are many different kinds of process element, there is a rich space for investigation. Possible kinds of element include those from traditional and agile processes, large and small projects, elements from process models, and

many more. As a means of structuring the evidence that represents the model's ability to capture and compare process elements, I have chosen an approach called *argumentation* along with an established notation for structuring arguments, *Goal Structuring Notation (GSN)*. *Argumentation* is "an approach which can be used for describing how evidence satisfies requirements and objectives" [160]. The use of a suitable notation such as *GSN* helps researchers to easily identify what evidence is required and helps stakeholders see at a glance what is the 'evidence coverage'. This approach has been used for many years in the safety critical domain and has recently been applied in the software domain [160].

The need to capture existing models that are the basis of various studies means that I must provide a means of representing studies that vary in integrity. One consequence of this is that it must be possible to capture models based on different beliefs, for example, beliefs about which contextual factors most affect outcomes. This will be necessary until the industry has progressed to a better understanding of these factors. This suggests that I must find an abstraction that accommodates a potentially huge variation in the statement of possible influencing factors. I am also required to capture processes that may have different kinds of objectives, for example, relating to cost or quality. For these kinds of reasons, the solution model will be realised as a framework, the elements of which are models in their own right. For example, there is a model (abstraction) for the contextual factors and one for the process objectives.

Evaluation of the framework will involve:

1. Identification of the range of processes and process models that must be successfully represented.
2. Discussion about how the framework meets the criteria established as a result of examination of process characteristics and process model limitations.
3. Presentation of evidence to support the claim that identified processes and models can be represented and that representation supports comparison.
4. Discussion about some limitations inherent in the approach.

1.6 Overview of Contributions

The main contribution of this thesis is the identification of the need for a holistic approach to modelling the software development process in a descriptive way and the presentation of a candidate modelling framework that provides a way of representing and comparing different kinds of process elements.

A second contribution is the identification of the various research groups that model the software development process to predict outcomes and the understanding of how these groups

differ in approach and what are the limitations inherent in the work of each. The contribution also includes a realisation that the narrow approach taken by each of the groups is a symptom of lack of real understanding and is the basis for the case for a more holistic approach.

A third contribution is the establishment of an approach for developing and evaluating models that claim to describe systems in a holistic and explanatory way. The strategy is to first establish model objectives and identify a comprehensive range of example systems to be described by the model. The next step is to examine the characteristics for, and problems with, the example systems to help identify key model properties. These provide some basis for establishment of a suitable model structure and may be used as criteria against which to evaluate candidate models in a preliminary evaluation step. Finally, the ability of the candidate model to satisfy objectives is established by accumulating a portfolio of different kinds of evidence relating to the example systems.

A final contribution is the understanding that the existence of a suitable framework gives rise to a number of unplanned research directions and the addressing of one such possibility, that of process-specific risk profiles. Remaining directions include the use of the framework to support research. Such directions are suggested as areas for future research.

1.7 Terminology

The issue of terminology in the field of software engineering is problematic. Words such as ‘task’ and ‘activity’ are used by different authors to mean the same thing. ‘Process’ and ‘product’ also tend to be undefined and many other terms are applied without any definition of what they mean.

My approach in this dissertation is to define terms used in a general way in a Glossary (see Appendix A). Such terms are *italicised* and defined on first use and subsequently italicised only when this helps clarify content. I also use *italicised* text when emphasising a word or phrase, even if not included in the Glossary.

For terms used by authors of a study being described, I include the term in ‘single quotes’. For example, a process might be described by an author as comprising a number of ‘Activities’ and ‘Tasks’. In such cases, I do not try to define exactly what the term means, unless this is necessary for the discussion.

I also use ‘single quotes’ when paraphrasing and “double quotes” when quoting phrases from other sources.

For elements of the model that is the subject of this thesis, I use *Slanting Text*.

1.8 Document Roadmap

This dissertation is placed in the area of *software processes* used by *projects* to produce software-intensive *products*. A *project* is “.. a temporary endeavour to create a unique service or product and with a definite beginning and end” [135]. I note that this definition says nothing about the form of the service or product delivery and, in this thesis, I view a project as any effort that makes a delivery of any kind to any stakeholder. For example, a project might deliver a finished product to a customer, a prototype to the development group or a test plan to the test group. In other words, project objectives and scope are decided by the interested parties and project definition is constrained only by the need to have a defined start and end and agreed delivery. A *software process* is “... the set of all activities which are carried out in the context of a concrete software development project” [59]. A *product* comprises the artifacts that implement a software-intensive system and are the deliverables from a project.

This thesis is organised as follows.

In Chapter 2, I consider some different viewpoints on what is software development and note that the range of proposed paradigms indicates a lack of real understanding of the essential nature of the activity. I then provide an overview of some common software development processes. I finally discuss some ways in which processes are categorised. I suggest that the interest in categorising is a symptom of lack of understanding and that focus should return to the more important task of understanding what are the key characteristics common to all software processes.

In Chapter 3, I examine the various kinds of study carried out by researchers with the goal of predicting software process outcomes. I learn that there are three separate communities and each applies a different approach and builds different kinds of models of the process. I expose some limitations inherent in the work of the different groups by showing that the kinds of models applied by each contain ambiguities and assumptions that render impossible comparison of results.

In Chapter 4, I examine research related to the goal of process flexibility. This related work spans several research areas, some directly related and others more indirectly. I first discuss processes and process frameworks for which claims of flexibility are made and suggest limitations based on an inability to capture the different kinds of process presented in Chapter 2. I then examine process models for which claims of flexibility are made and show how each is limited according to its underlying approach as discussed in the previous Chapter. I finally discuss attempts to model human-related factors and note that research is progressing in this direction but, as yet, no suitable model of the human aspects of the process exists.

In Chapter 5, I present a justification of the need for a theoretical model of the software development process. I first provide a general overview of the different approaches to gathering

data and discover that, if the aim is to establish causal connections between factors, a theoretical framework is required. I next present some quotes from a number of researchers stating the need for a holistic theory and reminding us that a characteristic of studies that are not based on such a theory is an inability to achieve consistent results. Finally, I suggest that current software process research is achieving inconsistent results because the research is fragmented and is not based on an underlying theory. I conclude that a formal model of the software process is required and formalise the objectives for such a model in the context of this thesis.

In Chapter 6, I analyse the various processes and process models described in Chapters 2 – 5 with the aim of understanding what might be the properties of a model that is a solution to the problem of process customisation. I also consider some ‘real-life’ scenarios from industry to help with identification of such properties. I then list the desired properties to be used as criteria against which to evaluate a candidate model.

In Chapter 7, I present a candidate model, *KiTe*. My approach to presentation is to first provide a schematic overview as a ‘gentle’ introduction, and to then present and formalise the abstract model.

In Chapter 8, I present some evidence to support the proposed model. This includes evidence to support the claim that *KiTe* may be used to capture any process or process model and that it supports comparison of studies.

In Chapter 9, I discuss how the existence of a suitable model provides some benefits not originally planned or realised. As illustration, I show how *KiTe* can be applied to the identification of risks specific to XP (eXtreme Programming) processes.

In Chapter 10, I evaluate the candidate model *KiTe*. I first discuss how well *KiTe* fulfils the criteria stated in Chapter 6. The aim of this discussion is to provide some preliminary confidence that *KiTe* will address the various process characteristics described in Chapter 2 and overcome the limitations of current process models described in Chapters 3 and 4. I then examine the evidence presented in Chapter 8 and discuss the strengths and weaknesses of the evidence. The studies that comprise this evidence represent attempts to find inadequacies with *KiTe* as a solution to the problem of capture and comparison. I find that the evidence is reasonably strong, but there are some serious gaps. I finally discuss some limitations inherent in the approach taken.

Chapter 11, I summarise the thesis and suggest some future research directions resulting from the research.

2

Software Development Processes

In Section 2.1, I present some different ideas about what is the nature of software development and suggest that the diversity of ideas is an indication that the software community lacks understanding of the essential nature of the software process. In Section 2.2, I overview a sample of software development processes, selected for their differences. In Section 2.3, I discuss some ways in which processes are categorised. I suggest that the interest in categorising is a symptom of the lack of understanding and represents an attempt to solve problems the industry does not understand. I also suggest that discussions and research based on polarisation do not help the industry to progress towards understanding what are the key characteristics common to all software processes and that focus should return to understanding these characteristics.

2.1 Nature of Software Development

In the 1960s, programming was largely seen as an *art* and most practitioners had received no formal training in the field [156]. As software systems became larger and more complex, this perception of the practice became less appropriate and researchers began to be concerned about the lack of a sound theoretical basis. As a result, they began to apply an approach based on manufacturing processes i.e. a process involving a single analysis, design and production stage.

One of the key events in the history of Software Engineering was the conference organ-

ised in 1968 by the *NATO Science Committee* to discuss issues relating to “software manufacture” [118]. The term “Software Engineering” was coined at this conference with the deliberate intention of implying the need for the discipline to be more strongly based on theoretical and engineering principles. It is interesting to note that nearly all of the issues believed to be of relevance today were brought up at this conference. For example, attendees discussed the need to iterate and obtain feedback from the customer, reuse of components in preference to continually ‘reinventing the wheel’, the lack of clarity about what ‘production’ means in a software project, measuring the production process, the risks involved in delivering software implemented using new and unproven techniques and the importance of finding a suitable abstraction for the software product. However, despite such discussions, there appears to have been little attempt to reframe the creation of software-intensive systems as anything other than a *manufacturing* process, with stages of analysis, design and production. This paradigm was compliant with the then-popular *waterfall* process model, in which each stage must be completed and fully documented and verified before the next one commences. One result of the use of this paradigm has been the application of *process control* principles to manage quality outcomes.

There has been much evidence in the form of failed projects to support the idea that a single pass *manufacturing* approach is not generally an appropriate one for software systems, and one result of this was the updating in 1987 of the US Department of Defense *DoD* standard *DoD-Std-2167* to replace the waterfall model with an iterative approach (*DoD-Std-2167A*). Despite evidence and change in standard, the industry has continued until recently to exhibit a “waterfall mentality” [95].

During the 1980s and early 1990s, various groups around the world recognised the need to deal with change and uncertainty relating to software project deliverables and, becoming frustrated with the unsuitability of the waterfall model for such projects, came up with methodologies of their own. These methodologies were seen as *lightweight* and were oriented towards frequent deliveries and feedback from the customer. This approach represents a paradigm of *software as a service*, and in 2001, representatives of these processes formed an *agile alliance* to promote the approach. Representative methodologies are termed *agile*, and include, for example, *XP*, *Scrum*, *Dynamic Systems Development* and *Feature Driven Development*.

Cockburn describes software development as a *cooperative game*, with the aim of the game being to make “ideas concrete in an economic context” and under limited resources. He appears to view ‘process’ as being a number of pre-defined sets of instructions aimed at removing dependence on key individuals but which actually overconstrain individuals. He advocates an agile approach [29] and specifically uses the *cooperative game* paradigm in his *Crystal* methodology.

A number of other paradigms have been proposed for software development although, to my knowledge, none of these has spawned representative methodologies. Ricardo Peculis also believes that a rigid engineering approach using process control techniques constrains developers

and reduces creativity and that people are the key. His solution is to regard software development as a *complex adaptive system* and claims that software management should be based on chaos theory along with leadership to orient the system to achieve expected results. No specific solution is presented other than a need to use “alternative, chaotic models” [127]. Lehman revisits the problem of uncertainty about what ‘production’ means in a software project, and declares software development to be a *design* activity [96]. Researchers from the discipline of psychology view programming as a “complex cognitive and social task ... involving several kinds of specialized knowledge” [128]. Armour believes software processes are about finding out what we do not know [6] and Curtis et. al. also have the view that software is a *problem solving* process and thus is at least partially about *uncertainty and learning* [37]. Several authors perceive software development as a *probabilistic* process, and this paradigm has been applied in efforts to model the process (see Chapter 3).

In recent years, there has been a revival of interest in the idea of software development as *craftmanship*. According to Pete McBreen, very large projects are generally hardware-driven and are really *systems engineering* projects, but most software projects require fast delivery and low cost, and the *craftmanship* paradigm is more suitable. In this paradigm, people are more important and “Software craftsmanship stands for a different kind of relationship between developers and users” [108]. DeGrace also mentions *craftmanship* [42] but believes that this viewpoint has in fact led practitioners to become almost religious in their attitudes, and we are “pulled this way and that by those who have the knowledge of The Right Way and The True Faith”. DeGrace also mentions the *hero* paradigm for software development and includes in his book quotes from several individuals who have single-handedly produced successful software, for example, Bill Gates and Andy Hertzfeld. Dawson et.al. suggest that the large variation in project circumstances mean that “guiding principles are hard to establish” and this has led to the belief that software development is an art or craft. The result is “individuals forming their own ideas for working practice based on a mixture of their own experiences, hearsay from others and general folklore and myths” [38].

Kitchenham and Carn, after consideration of what is the practice of software engineering, conclude that “the software production process is an engineering discipline like any other engineering discipline” [88]. As support for this claim, I present three definitions of *engineering*. The American Heritage Dictionary of the English Language defines it as “The application of scientific and mathematical principles to practical ends such as the design, manufacture, and operation of efficient and economical structures, machines, processes, and systems” [63]. From Wikipedia, “Engineering applies scientific and technical knowledge to solve human problems. Engineers use imagination, judgment, reasoning and experience to apply science, technology, mathematics, and practical experience. The result is the design, production, and operation of useful objects or processes” [163]. Hansen believes engineering is “...the systematic ap-

plication of scientific knowledge in creating and building cost-effective solutions to practical problems” [43]. It would appear that the engineering paradigm is an appropriate one for the problem of software development. However, it is clear from the discussions above that this term is viewed by some as relating to only large systems, and Hansen points out that the concept of applying ‘Engineering’ to software still arouses debate [43]. Perhaps this also is a consequence of the original usage of the term at the NATO Conference being associated with ideas of large, critical systems and a manufacturing mindset.

One interesting point that arises from the above discussions is that most authors equate *process* with *document-driven, manufacturing process*. However, a *process* may be defined simply as “The sequence of activities, people, and systems involved in carrying out some business or achieving some desired result” [69], and I would argue that models that represent all of the above paradigms can be represented as a *process* according to this definition.

In the above discussion, I presented many different ideas about what is software development. The lack of consensus within the industry is, in my view, a clear indication that the space is not yet well understood. The processes presented in the next Section are each founded on a particular idea and represent the industry’s solutions to an unstated and vaguely understood problem. This view is supported by Fuggetta, who believes that “the approach of most Software Engineering researchers is oriented to inventing new ‘things’ rather than pursuing a deeper understanding of the problem we want to solve . . .” [54].

2.2 Processes Overview

The processes overviewed in this Section are selected for variation in characteristics. Some common processes excluded are *Adaptive Software Development (ASD)*, *Dynamic Systems Development (DSDM)* and *Lean Software Development*.

2.2.1 Waterfall

A *waterfall* process involves a number of sequential phases, for example, ‘Gather requirements’, ‘Design’, ‘Implement’ and ‘Test’. Each phase includes a verification step, and the phase should be completed and fully documented before the next begins. As in a manufacturing process, it is most likely that each phase will involve different people, for example, analysts, designers, coders and testers, and so communication is based on large amounts of documentation. Feedback between phases is permitted, but this is very controlled and no phases are omitted. For example, if problems in requirements are discovered during test, all of requirements, designs, code and test artifacts should be updated to reflect the change.

In a ‘pure’ waterfall process, a single delivery is made to the customer i.e. there is no opportunity for refining of requirements from customer feedback. In a real software development project, this approach is impractical [142]. Seldom is the end customer sure at the start of the project exactly what he or she wants and so a constant stream of change requests must be managed. Maintaining consistent documentation is difficult under time pressure and documentation tends to become inconsistent between phases. There is a tendency for developers to start work on subsequent phases before earlier ones are complete. It is doubtful that many projects have in fact implemented a ‘pure’ waterfall process [95] and there are many reports indicating a more iterative approach in practice.

Winston Royce is often quoted as creating the waterfall model in 1970 [141]. However, according to Boehm, the waterfall model is a refinement of a stagewise model introduced in 1956 to address problems of inadequate architecture and difficult-to-read code resulting from the ‘code-and-fix’ approach practiced in the earlier years of programming [23]. In fact, Royce has been somewhat misinterpreted, perhaps as a result of the ‘manufacturing mindset’ existing at the time, as he includes in his documented process the steps “do it twice” i.e. deliver the second version as final version to the customer and “involve the customer” i.e. solicit feedback from the customer throughout the process [141].

The industry has spent much effort in debating the usefulness of a waterfall approach to software development and it might be argued that the majority of subsequent processes have been created in reponse to the inadequacies of the waterfall model. Some claim that the waterfall approach “pushes risk forward in time so it is costly to undo mistakes from earlier phases” [92]. It has also been noted that the waterfall represents an attempt to *manage* the development of software and as such is not based on an understanding of the processes involved and so does not reflect the real activities that take place [88]. This means that it does not capture cause-and-effect relationships and so cannot be used as the basis for a model for describing different kinds of process.

2.2.2 Spiral

The *Spiral* model of software development was created in 1988 by Barry Boehm as a *risk-driven* approach to the software development process [23]. The model was a response to the belief that the then popular waterfall approach was “discouraging more effective approaches to software development such as prototyping and software reuse”. The reason for this is that the waterfall defines a strict order for process phases and so, for example, does not account for situations in which the coding of a prototype is appropriate before requirements are consolidated.

The model comprises a number of cycles, the required number of these varying with the project. Each cycle commences by determining cycle objectives (what the cycle must achieve),

alternatives (possible ways to achieve objectives, for example, reuse) and constraints on the alternatives (for example, cost, module interfaces). The next step in a cycle involves performing a risk analysis on the possible alternatives to meet the objectives. The third step is to create and test the deliverables required to meet the cycle objectives and the fourth to plan a subsequent cycle, if required. For some projects, objectives for early cycles might be high level in nature and involve, for example, feasibility studies or prototypes as deliverables. The cycles would then begin to address more specific issues, for example, requirements and designs. Smaller, more well-defined projects might commence with a ‘formal requirements’ cycle. A key point is the inclusion of a risk identification and management phase during every cycle.

The spiral model provides much flexibility, as what is to be achieved during each cycle is defined by those using the model. This means that the model encompasses other models, for example, waterfall and it also means that other models can be combined within the spiral framework. For example, if cycle objectives are defined to be consistent with those of the waterfall phases, the spiral becomes a waterfall model. In a similar way, a project with high risk of creating the wrong user interface and with low budget risk might implement many ‘requirements/design/code’ cycles, and risk considerations might lead to a decision not to document specifications. The spiral thus becomes equivalent to an *evolutionary* approach in which content is allowed to evolve as the project progresses.

Boehm acknowledged a number of difficulties in applying the model. These include the need for greater process determination when software development is contracted out, the need for risk-assessment expertise and the need for further elaboration of steps [23]. I note that the approach is a response to the inability of waterfall to accommodate situations in which different phasing is appropriate.

2.2.3 Rational Unified Process (RUP)

The *Rational Unified Process*, (*RUP*) was developed by *Rational Software* and integrated with its suite of software development tools [92]. Its base is the *Objectory* process created in 1987 by Ivar Jacobson, a process centred on the concept of use case and object oriented design and obtained on merger of Rational with *Objectory AB* in 1995. The addition of the *Booch Method*, an iterative approach to OO analysis and design, requirements management (from *Requisite, Inc.*) and a test process (from *SQA, Inc.*) resulted in the *Rational Objectory Process*. RUP resulted from further mergers, resulting in acquisition of business modelling, project management and configuration management capability. I observe that elements included originally related specifically to software development activity, and have extended over time to include software development support and project management.

The process applies to development efforts that use an *Object Oriented (OO)* approach.

Processes ('workflows' in RUP) are included for project management, business modelling, requirements, analysis and design, implementation, test, configuration management, environment and deployment. These span a number of 'phases' and iterations are encouraged within each 'phase'.

RUP is an artifact-driven process and 'workflows' are constrained by the tools supported in the product. It is possible to configure the process by modifying steps and adding guidelines and checkpoints, but within the provided structure of provided 'phases', 'iterations' and 'workflows'. Key ideas include the need to manage risk and to establish a core architecture during early iterations [95].

RUP does not define a single *process*, rather it is a *process framework*. The claim is that the RUP framework represents an attempt to bring together known best practices for software development and project management. The idea is that users of RUP create processes based on the included 'best practices'. However, the claim of supporting 'best practices' is tenuous and Rational fails to justify such claims by reference to available evidence. Indeed, a 'best practice' appears to be defined as one currently supported by RUP. For example, RUP documentation references the *Software Program Managers Network (SPMN)* [75] as the source of project management best practices. However, the interpretation of these practices in RUP is loose. Two of the six RUP best practices are 'Develop Software Iteratively' and 'Visually Model Software' but these practices are not included in those suggested by SPMN. The SPMN practices of 'Adopt Continuous Risk Management' and 'Use Metrics to Manage' are not included in the RUP list, but rather are catered for as specific aspects of the 'Project Management Workflow'.

2.2.4 Cleanroom

The *Cleanroom* approach to developing software was developed by Harlan Mills and was initially practiced in 1987 at IBM [150]. The traditional, craft-based approach of the time viewed the introduction of *defects* into the software and the related costs of detection and removal as inevitable [100, 150]. Management focus was on moving into execution quickly in order to commence debugging. The introduction of a phased, waterfall approach represented an attempt to control software quality by applying a manufacturing-based process. In a manufacturing situation, items found to be defective at the end of the process are discarded. The aim is to reduce the number of defective items by improving the process. However, for software-intensive products, the concept of discarding defective items does not apply, rather defects are repaired prior to delivery. For a manufacturing-based process such as waterfall, the cost of repair for defects discovered at the end of the process is high because of the large quantities of documentation involved.

Mills recognised that defect removal at the execution stage is an inefficient activity and

the key idea of the Cleanroom approach is one of defect prevention. “Cleanroom software engineering is a set of principles and practices for software management, specification, design and testing that have been proven effective in improving software development quality while at the same time improving productivity and reducing cost” [41]. The name ‘Cleanroom’ was taken from the electronics industry, and represents the vision of zero defect injection. Studies report that the numbers of defects found in code as measured prior to initial unit test activity is greatly reduced by the application of a Cleanroom approach and the kinds of defects discovered are generally “... simple mistakes easily found and fixed ...”.

The approach represents a move “from traditional, craft-based software development practices to rigorous, engineering-based practices”. The theoretical foundations of Cleanroom are formal specification and design, mathematically based correctness verification and statistical testing. The process involves small teams developing and certifying software increments, with a hierarchical arrangement of teams for large projects. System integration is continuous and developers maintain intellectual control [100]. Developers do not execute and test their code, rather independent teams carry out all verification and testing from first execution. The aim is to quickly deliver an initial product of high quality and then incorporate new requirements. Cleanroom prototypes are used to elicit feedback when requirements are unclear.

Specifications are generally developed by development and certification teams working together with the customer. Functional and usage scenarios are defined and include both correct and incorrect examples. The functional specification forms the basis for development and the usage specification for the generation of test cases. Specifications are used for increment planning. Developers carry out design and correctness verifications for each increment using the concept of ‘box structures’. When an increment is completed, it is integrated and delivered to the test team who execute test cases. Testing is viewed as a statistical experiment i.e. a “...representative subset of all possible uses of the software is generated, and performance of the subset is used as a basis for conclusions about general operational performance. In other words, a ‘sample’ is used to draw conclusions about a ‘population’.” [150] If quality standards are not met, testing ceases and developers return to the design stage.

Claimed benefits of the approach include significant improvements in correctness, reliability and understandability. These are supported in a 1987 empirical study by Selby, Basili and Baker, who found that Cleanroom teams met deliveries more frequently than non-Cleanroom teams and produced code that contained fewer defects and was of higher quality. Eighty-one percent of the Cleanroom developers said they would use the approach again [147].

Limitations of the Cleanroom approach include the requirement for training — managers must have a “sound understanding of Cleanroom imperatives” and developers must be sufficiently skilled to “adapt the process to the local environment” [150]. It is also believed to be ineffective to use Cleanroom to effect small changes to software developed using non-Cleanroom

technologies. Other limitations for some projects include the difficulty of defining a “representative subset” of uses for testing, the high cost of implementation and the need for independent teams.

2.2.5 Agile alliance

In the 1990s, several individuals who were unhappy about the use of traditional methods used for creating software independently evolved processes they believed to be more appropriate for ‘real’ software projects. From Europe emerged *Dynamic Systems Development Methodology (DSDM)*, from Australia *Feature-Driven Development* and from the USA *Extreme Programming (XP)*, *Crystal*, *Adaptive Software Development* and *Scrum* [168]. Despite the fact that the methodologies appeared to have little in common, representatives from each met in 2001 in an attempt to find common ground and the ‘Agile Manifesto’ was formed. The common ground was that participants were “sympathetic to the need for an alternative to documentation driven, heavyweight software development processes” [61]. The resultant manifesto represents an attempt to redress a perceived process-heavy imbalance in the industry by adopting the philosophy that people play the key role in software development and process must play a secondary role [32]. I overview some of the agile methodologies and discuss further in the next Section.

2.2.6 XP

XP (*eXtreme Programming*) is a “light-weight methodology for small-to-medium-sized teams developing software” and was proposed by Kent Beck as a response to the need to manage “vague or rapidly changing requirements” [15]. Beck introduces four ‘Values’ and a number of basic ‘Principles’ that realise the ‘Values’. His process solution comprises the development ‘Practices’ that comply with the ‘Principles’.

The Practices cited by Beck are [15]:

The Planning Game Quickly determine the scope of the next release by combining business priorities and technical estimates. As reality overtakes the plan, update the plan.

Small Releases Put a simple system into production quickly, then release new versions on a very short cycle.

Metaphor Guide all development with a simple shared story of how the whole system works.

Simple Design The system should be designed as simply as possible at any given moment. Extra complexity is removed as soon as it is discovered.

Testing Programmers continually write unit tests, which must run flawlessly for development to continue. Customers write tests demonstrating that features are finished.

Refactoring Programmers restructure the system without changing its behavior to remove duplication, improve communication, simplify, or add flexibility.

Pair Programming All production code is written with two programmers at one machine.

Collective Ownership Anyone can change any code anywhere in the system at any time.

Continuous Integration Integrate and build the system many times a day, every time a task is completed.

40-Hour Week Work no more than 40 hours a week as a rule. Never work overtime a second week in a row.

On-Site Customer Include a real, live user on the team, available full-time to answer questions.

Coding Standards Programmers write all code in accordance with rules emphasizing communication through the code.

In this paradigm, working software is delivered every couple of weeks and the above Practices carried out for each cycle. Rather than defining the product up-front, the approach is to allow it to grow in an evolutionary way, as customers become more clear about what is wanted as a result of feedback. Beck suggests that the XP Practices support each other and that process efficacy will be compromised if any are missing.

2.2.7 Scrum

Scrum was developed by Ken Schwaber in 1996 and is based on the notion that software development is inherently unpredictable. The mitigation strategies for the ‘unpredictability’ risk factor include 30-day work intervals and a daily status meeting of developers, customers and managers. Developers work from a prioritised list of features. Key principles are [30]:

- Small working teams that maximize communication, minimize overhead, and maximize sharing of tacit, informal knowledge.
- Adaptability to technical or marketplace (user/customer) changes to ensure the best possible product is produced.
- Frequent ‘builds’, or construction of executables, that can be inspected, adjusted, tested, documented, and built on.
- Partitioning of work and team assignments into clean, low coupling partitions, or packets.
- Constant testing and documentation of a product as it is built.
- Ability to declare a product ‘done’ whenever required.

No guidelines are given on how to create the product. I suggest that this methodology represents a management approach to risk mitigation.

2.2.8 Crystal

This is a family of methods created by Alistair Cockburn to address the problem of poor communication in software projects. All methods have a core set of ‘roles’, ‘work products’, ‘techniques’ and ‘notations’ and the set expands as the team size grows or priorities change. Priorities are project dependent, for example, ‘productivity’, ‘system criticality’, ‘legal liability’ [30]. Practices from both agile and plan-driven methods are implemented and techniques from psychology and organizational development research incorporated [21]. All crystal methods have three ‘Priorities’. These are ‘project outcome’, ‘efficiency’ and ‘habitability’. Shared ‘Properties’ include ‘Frequent Delivery’, ‘Reflective Improvement’ and ‘Close Communication’ [53].

2.2.9 Feature Driven Development (FDD)

FDD was developed by Jeff DeLuca and Peter Coad as the result of an attempt to save a failing project [30]. This is an architecturally based process in that an overall object architecture is established up front along with a features list [21]. Features are “small items useful in the eyes of the client” [30]. They are captured in a language understandable by all parties and each is expected to take no more than 10 days to develop. The role of Chief Architect and Chief Programmer are maintained and OO design methodologies implied. Adaptation is achieved by 2-week iterations. FDD does not mandate daily involvement of the client and a central repository is used to capture all important project information, for example, minutes, knowledge, decisions and issues.

2.3 Categorising Processes

Many practitioners and researchers spend much time in the ‘traditional versus agile’ debate [16, 21, 30]. Terms such as ‘plan-driven’ are used to describe traditional processes, with the inference that agile methods are weak as regards planning [16]. Others believe agile processes exhibit strong planning, as the setting of customer expectations is supported by short cycles and estimation and monitoring are inherent aspects of agile processes. Articles state that agile software development “is about feedback and change” [168], implying that traditional methods are unsuited to projects displaying these characteristics. Larman and Basili point out that iterative and incremental practices to specifically address problems of change have been carried out from at least 1968 [95].

There is general agreement that the actual practices implemented in agile methodologies are not new. Although many authors have attempted to characterise agile methodologies [5, 16, 21, 30, 168], confusion reigns as to what exactly are the distinguishing characteristics. In

the overviews presented above we see a range of combinations of practices with ‘traditional’ and ‘agile’ practices interspersed. For example, daily customer contact is a recognised ‘agile’ practice but is not implemented in FDD; up-front features definition is a recognised ‘traditional’ practice but is practised in Scrum. Fowler believes there are “some fundamental principles that unite these methodologies [53]”. He believes that the agile approach of minimal documentation is a symptom of two deeper characteristics and these are a belief in the adaptive rather than predictive nature of software development and an orientation towards people as key to success. As support for the former, he cites Cockburn who points out that process predictability requires linear components but people are not linear. The latter means that the role of a process is to support the development team [53]. Both characteristics proposed by Fowler confirm the people-centric nature of software development.

Although processes are commonly categorised as ‘traditional’ or ‘agile’, I notice that other kinds of categorisation might be applied. If I categorise according to ‘what is software development?’, I find Waterfall and Cleanroom are categorised as ‘manufacturing’, Spiral as ‘risk management’, XP as ‘service’, Crystal as ‘cooperative game’, and so on. I also notice that some paradigms have no representative process. For example, Curtis’s plea for a behavioural model of the process has not been actioned and no representative process exists.

A third possible categorisation involves determining the degree of definition of a process. I notice that XP is very tightly defined. Iteration length is fixed, rules for how to design and code are very explicit (as it is expected that techniques such as *pair programming* and *common code base* will be implemented) and communication strategies are mandated. Spiral, on the other hand, provides complete flexibility in how to create the product and mandates only that the management practices of planning and risk identification be carried out in a specific way. Decisions about the use of prototypes, how to design and code and when and how to inspect are left to the user of the process. Waterfall also leaves many decisions to the process user. Although based on a manufacturing, and thus tightly defined, process, opportunities for flexibility result from the facts that people perform the process tasks and that defects are generally resolved rather than thrown away. Decisions about, for example, coding techniques and inspection and rework policies are left to the process user and it is the phasing only that is defined for Waterfall.

The problem space is a rich one and processes can be categorised in several ways. However, I suggest that such categorisation only succeeds in confusing the issue by polarising processes in a way that makes it difficult to understand the common factors. For example, any of the available techniques that ‘characterise’ agile processes, for example, documentation, up-front requirements, test-first design, status meetings, iterations and customer collaboration, can be, and have been, applied in both agile and traditional situations. Situations that appear on the surface very different may be viewed as different solutions to the one problem. For example, managing product definition may be achieved by exhaustive up-front capture of requirements

with a single delivery of the product, up-front capture with incremental delivery allowing for some feedback from the customer or minimal initial product definition and many prototypical deliveries aimed at growing the product in an evolutionary way. The first is believed to be appropriate for products such as compilers and the last for products with low understanding of user interface requirements [23]. A similar discussion around how product knowledge is held leads to the identification of ‘suitable’ and ‘risk’ situations. If knowledge is held in documents, it is believed that developers have a view of only part of the product, with little understanding of the rationale for the product or the characteristics of the application area. If knowledge is in peoples’ heads, there is a risk of knowledge being unavailable, for example, if developers leave or are unable to share knowledge for any reason. The point is that, for a given project, the ‘best’ strategy will depend upon a consideration of specific solution strategies in the context of the project. There is no need to be ‘agile’ or ‘agile with a little traditional’ or, maybe, ‘traditional with a little agile’.

In summary, I suggest that, if the goal of customisation is to be met, discussions that categorise processes in a polarising way are unhelpful and focus needs to be on understanding what are the characteristics of the software process and how these relate to specific project contexts. I provide some support for this viewpoint in Section 10.3.

3

Software Process Models

In an attempt to better understand the issues involved and thus facilitate prediction capability, researchers use various techniques to model the development process. There are three main groups involved in modelling for understanding and prediction, each with a different kind of goal. Researchers in the first group apply statistical manipulations on existing datasets with a view to predicting outcomes on future data sets. Models based on statistical prediction techniques are overviewed in Section 3.1. Researchers in the second group are involved in formal experimental research with the aim of providing sound research data for use in further studies. Some research based on this approach is overviewed in Section 3.2. Researchers in the third group model and simulate the development process, often with the aim of perturbing the process to study what effect this has on specific outcomes, for example, quality. Some of these models are overviewed in Section 3.3.

One characteristic all groups have in common is that the lack of available, sound data constrains efforts in some way with the result that models often contain ambiguities and are based on unstated assumptions. This means that we do not really understand what is the meaning of results achieved. For each of the groups, I identify the kinds of limitations characteristic of the kinds of models created.

3.1 Predictive Modelling

3.1.1 Cost estimation

In the 1960s and 1970s, researchers and practitioners became concerned that software-intensive projects were plagued by problems not present in other manufacturing projects. These problems resulted in failure to meet delivery expectations and spawned a number of models aimed at supporting predictions of project cost and duration [103]. The models in general represent a pragmatic approach aimed at improving estimation accuracy rather than an attempt to understand underlying causes. To this end, models apply, for example, statistical methods to predict based on existing data.

Early predictive models aimed to facilitate the prediction of costs and durations for a given project [19, 103]. These cost estimation models are equations linking costs to the size of the software product to be delivered and a number of other factors believed to influence costs, for example, staff size. The form of the model equation is inferred from a statistical manipulation on a number of datasets collected from real projects. Cost estimation activity has continued to the present time and different kinds of techniques applied.

Possibly the earliest known model is that of Farr and Zagorski, introduced in 1965. The model has thirteen predictors and estimates manpower required from delivery of complete requirements to release for integration i.e. for design, code and debug [103]. The Wolverton model from TRW Systems (1974) assumed manpower is directly proportional to size and uses historical data, a phased approach and a ‘difficulty’ scale for old or new software. The Doty model (1977) is a set of recommendations for estimating and may be applied to command and control, scientific and business systems. Size is used to compute a cost, which is refined using seventeen environmental predictors [103]. SLIM was developed in the late 1970s by Larry Putnam. This model uses Source Lines of Code (SLOC) for project size and then modifies this through the use of a Rayleigh curve model to produce effort estimates. Two key parameters influence the shape of the curve — the ‘manpower buildup index’ (initial slope) and ‘productivity factor’ [25]. The COConstructive COst MOdel (COCOMO) was developed by Barry Boehm of TRW and is based on an analysis of 63 software development projects. The model predicts effort and duration based on size measured in KDSI (thousands of delivered source instructions) and a number of ‘cost drivers’. There are three alternatives for model equations for different kinds of project.

More recent studies consider a range of data-intensive modelling techniques such as ordinary least-squares regression (OLS), Analysis of Variance approach for unbalanced data sets (ANOVA), classification and regression trees (CART), analogy-based approaches [7, 25] and data pruning approaches [28].

Practitioners and researchers continue to express concern about their inability to accurately estimate costs based on the above models [77, 82]. A 1987 empirical evaluation of four popular models, SLIM, ESTIMACS, Function Points and COCOMO revealed that when models were applied in an environment different from that in which the model had been developed, average error rates $((\text{estimated effort} - \text{actual effort}) / \text{actual effort})$ ranged from 85 to 772 percent. After calibration to the local environment, SLIM showed an 88 percent correlation between estimates and actuals, with other models performing less well. Most models appeared to be adding extraneous information [82]. A 1999 study by Briand et. al. examined a number of data-intensive modelling techniques using a large database of business applications. Techniques included the four modelling techniques introduced earlier in this paragraph. Results showed that outputs from all of the models was “from a practical perspective, far from satisfactory for cost estimation purposes” [25]. Researchers continue to investigate techniques for improving predictions. For example, Auer and Biffel propose an extension to an analogy-based approach i.e. one in which estimates are derived from historical data by finding projects with similar features. The extension takes into account the fact that different project features “are known to have varying impact on actual project similarity” [7]. They propose a scheme for weighting features based on relative impact. Chen et. al. observe that real world data sets “often contain noisy, irrelevant, or redundant variables”. They claim “huge improvements” if data for similar projects only is included and if most of the columns (i.e. input parameters in the data set) are pruned away [28]. These research efforts indicate that, although many agree that predictive models are useful under certain circumstances, care must be taken when selecting source data for predictions and models may not be applied in a general way.

3.1.2 Fault prediction

A second example of the use of statistical modelling techniques on large datasets concerns the various research efforts aimed at identifying what are the factors that affect the incidence of software *faults* [12, 18, 50, 58, 60, 83, 94, 112, 116, 123, 124, 172, 173]. This research is driven by the high cost of finding and fixing faults just prior to a product’s release. The strategy is to identify fault-prone modules earlier in the development cycle, for example, after design or just prior to testing, and thus enable development and testing efforts to focus on these high-risk modules.

This research has spanned twenty years. Inputs to the statistical models generally include a number of product metrics, for example, size measured as ‘lines of code’ or ‘Halstead program length’, control flow structure measured as ‘McCabe cyclomatic complexity’ and design coupling measured as ‘fanin’ and ‘fanout’ [94]. Models aim to either predict expected module fault density [12, 60, 58, 123] or to classify modules as ‘fault prone’ or ‘not fault prone’ [83, 94].

A number of statistical techniques has been applied to the problem. Early models applied multiple regression techniques but this approach was believed to be problematic because of the non-normal nature of the fault data (for example, modules typically exhibit no, or few, faults at later stages in the software cycle) [116]. Khoshgoftaar et. al. in 1996 applied a non-parametric discriminant analysis technique to classify programs as ‘low-fault’ or ‘high-fault’. Eleven complexity metrics were developed as independent variables, including ‘code lines’, ‘character count’, ‘Halstead’s program length’ and ‘McCabe’s cyclomatic complexity’. The problem of multicollinearity in the independent variables was addressed by applying principal component analysis, resulting in only two orthogonal (uncorrelated) complexity domains. Correlations between these two domains and program faults was found to be high [116]. Lanubile and Visaggio used techniques of principal component analysis, discriminant analysis and logistic regression to classify modules. They believe logistic regression is preferable to discriminant methods because the technique is not based on normality assumptions [94]. Fenton and Ohlsson remind us that data are measured on different scales and statistical analysis techniques must take this into account [50]. Graves et. al. apply ‘generalized linear models’ to determine fault rates as these are appropriate for non-Normal distributions, but comment that the “choice of parametric family ... led to some complications” [58]. Ostrand et. al. developed a negative binomial regression model to sort files in descending order of predicted fault density and report accurate predictions [18, 124].

3.1.3 Limitations

Cost estimation models have proved disappointing for estimating outcomes for real projects. One possible reason is that models are based on an assumption about what are the factors, in addition to size, that most influence productivity. COCOMO measures size in ‘Lines of Code’ (LOC) and includes, for example, ‘personnel experience’, ‘personnel continuity’, ‘database size’, ‘required reusability’, ‘virtual machine volatility’ and ‘requirements volatility’. Briand et. al. measure size in ‘Experience Function Points’ (derived from the Albrecht Function Point measure) and include ‘organization type’, ‘application type’, ‘customer participation’, ‘requirements volatility’ and ‘team skills of staff’. The implication is that the relevant factors are not yet properly understood. This creates a problem that is compounded by the difficulty of collecting data on these factors-of-interest. This idea is supported by the Briand study where comparisons between the use of company-specific data and multi-organisation data failed to show a significant difference when company-specific data was used. The authors suggest that “the main source of heterogeneity may come from the project characteristics themselves rather than the organizations where they take place” [25].

In the case of fault density prediction, there is some lack of agreement about results. An

example is the use of statistical techniques to predict the relationship between module size and fault density. Are small, medium or large modules most fault-prone? Several researchers have found that, contrary to popular belief and the notions of structured and modular programming, fault density has been found to increase in smaller modules and remain constant for increasing size [12, 112, 123]. Explanations given relate to the increase in interfaces for smaller modules. Hatton, however, makes the case for a U-shaped result and cites authors in addition to himself who have found that fault density increases as modules become smaller or larger than some minimum value[60]. His explanation involves the idea that humans hold a predefined number of pieces of information in short term memory — this corresponds to the ‘dip’ in the curve — and when this is exceeded i.e. the module is too big, long term memory mechanisms come into play. More recently, Fenton found no relation to module size and this was backed up in a later paper by Ostrand and Weyuker[124]. Lanubile et. al. compare several modelling techniques for predicting software quality by building models based on software product measures and using the models to classify components as high- or low-risk. Techniques included principal component analysis, discriminant analysis and logistic regression. They conclude that “no model was able to effectively discriminate between components with faults and components without faults”. They warn that, although past research has indicated correlation between product measures and fault densities, “the underlying phenomena continue to be poorly understood” and researchers are working with assumptions [94].

In all of the above examples, researchers use statistical techniques on existing datasets with a view to identifying influential factors for use in predictive models. Basic statistical wisdom tells us that, in this modelling paradigm, results are not applicable to situations other than those existing during data collection [31]. Statistical methods can show correlation but without understanding how factors in different situations might ‘change the rules’, there is no possibility of applying results to other circumstances. One symptom of this is that researchers find it difficult to obtain consistent experimental results, and this fact is reflected in the above discussion. The problem is acknowledged by a number of researchers in the field of software engineering. Lanubile et. al. believe that “Predictive models are very attractive to build, but they can be a waste of time if we rely on false assumptions . . .” [94]. The need for a deeper investigation into the underlying processes is reiterated by Zhang et. al. who believe that a failure to consider other factors that might contribute to software reliability has “become somewhat a limitation of the existing software reliability models” [173]. Fenton et. al. remind us that statistical models do not capture causal relationships and “recommend more complete models” that include explanatory factors [50].

3.2 Controlled Experiments

In 1976, NASA's Goddard Space Flight Center (NASA/GSFC) created a partnership with the University of Maryland (UM) and Computer Sciences Corporation (CSC) "for the purpose of understanding and improving the overall software process and products that were being created within the GSFC Flight Dynamics Division" [117]. This group is the *Software Engineering Laboratory (SEL)*. The strategies applied by this group involved measurement and experimentation. This application of controlled experimentation techniques to increase understanding reflected the desire of some researchers to increase the integrity of empirical research in the industry. However, perhaps because of the complexity of the software development environment, the approach has been adopted by only a small group of researchers.

It is, however, generally agreed by researchers that, if we are to progress as a professional discipline, it is now time to move away from the 'analytical advocacy research' [49] with which the industry is familiar and towards a more formal approach to experimentation. A recent (2005) literature survey examining controlled experiments in software engineering uncovered the fact that "the majority of published articles in computer science and software engineering provide little or no empirical validation and the proportion of controlled experiments is particularly low" [149]. Basili et. al. believe that "...in software engineering, the balance between evaluation of results and development of new models is still skewed in favor of unverified proposals" [14].

The industry is now witnessing an increased interest in sound empirical research. The standard techniques for empirical research include observational studies, for example, case studies, and controlled experiments [44, 90]. The former are generally carried out when the aim is one of exploration or comparison i.e. the researcher wishes to more fully understand some aspect of the system under study, perhaps as a preliminary step towards theory-building. Controlled experiments aim to examine causal relationships between various factors and study a problem stated as a hypothesis based on some theory or model [55]. Key aspects of this paradigm include an operationalisation that states what are the real-world entities that will be measured to represent the hypothesis, the soundness of the experimental design, control of any variables that might affect results, the use of random data and the application of appropriate statistical techniques for rejection of the null hypothesis.

Research into the software process can be categorised as examining the inter-relationships between process, product and people. For example, "Which inspection technique is better?", "Did the technique yield better results if the developers were experienced?". Some controlled experiments that have been carried out include studies examining various reading techniques for inspections and object-oriented designs and code [2, 117] and studies on regression testing [80].

3.2.1 Limitations

Formal empirical research tends to be careful about defining what is being measured and tested, but the problems with interpreting results is acknowledged by a number of researchers. Carver et al. [27] point out that there is a problem with understanding what are the common assumptions arising in current empirical research efforts. It is difficult for researchers to be sure that all possible explanations for results have been identified and that effects are, in fact, due to the cause under investigation. This is a problem of *internal validity*. For example, in experiments involving process and product, are we certain the human factors were held constant? According to Kitchenham et. al., "... controls are difficult to define because software engineering tasks depend upon human skills" [90] and an existing ontology of context "identifies a very large number of factors but does not offer any advice as to which factors are most important" [90]. We remember to take account of experience and skills, but are there any other factors that might confound results, for example, motivation and ease of communications? Carver et. al. comment that "the variation among the subjects can outweigh the influence of the real variable of interest" [27]. The "potentially large number of context variables" [14] also causes problems of *external validity* and "we cannot a priori assume that the results of any study apply outside the specific environment in which it was run" [14]. Sjoberg et. al remind us that "there is no generally accepted set of background variables for guiding data collection ... because the software engineering community does not know what are the important ones [149].

3.3 Simulation Modelling

A third set of researchers originally addressed the problems exhibited by software projects by attempting to enforce greater control of the process. The models created by this group were *prescriptive* in nature i.e. defined what steps were to be carried out and how. Such models tended towards automation of the process environment [67, 111]. This trend was highlighted by a well-quoted and much acclaimed keynote speech at the 1987 International Conference on Software Engineering (ICSE9), where Leon Osterweil declared that "software processes are software too" [122]. Others disagreed with this view and, at the same conference, Lehman delivered a 'response' paper claiming that "the existence of a programming language sets up constraints as to how a problem may be solved, severely limits human creativity" and that it "... is the problem domains ... that become well understood and formally modelled, not the the process for program development in general [97]". He believed that process programs "do not ... appear to provide a fundamental contribution to the further development of a software engineering discipline" and suggested that the challenge of the future was to improve clarification and understanding of the general process. He and many others have taken up this challenge by

creating executable models of the software process specifically aimed at understanding what are the key influencing factors [19, 81, 103, 130, 134, 137].

One of the early models in this direction was that of Abdel-Hamid and Madnick, based on a system dynamics paradigm (see Section 3.3.1). This model addresses the managerial aspects of software development [1] and represents a response to the idea that there was too much focus on modelling the technological aspects only of the process. Substantial attention is now given to modelling in a descriptive, rather than prescriptive, way and the term *simulation modelling* has been coined. An annual Workshop, the Workshop for Process Modelling and Simulation (PROSIM), is held in conjunction with the International Conference for Software Engineering (ICSE) and the Journal of Systems and Software has produced a number of issues dedicated to this subject.

Model simulations have used techniques from various disciplines and the scope of work has varied from small portions of the product lifecycle to long term organisational matters. The discipline is “increasingly being used to address a variety of issues from the strategic management of software development, to supporting process improvements, to software project management training” [81]. The main paradigms currently applied to simulation modelling are system dynamics [1, 51, 99, 105, 119, 138, 162], discrete event simulation, state based [67, 137], and rule based [46, 143, 153]. More recently, researchers have combined paradigms in an attempt to overcome limitations inherent in individual paradigms [45, 93, 107, 106, 136].

Modellers tend to approach a problem from the viewpoint of the paradigm selected. Lehman, Ramil and others have studied long term product evolution using a system dynamics approach [99, 138, 162] based on Jay Forrester’s work on the study of social systems [52]. Abdel-Hamid and Madnick have applied the system dynamics method to study manpower and quality-related issues [1]. A number of researchers have based their work on this. Pfahl and Lebsanft have used an extended model to study planning and control at the project level; several papers at Prosim 2003 applied a system dynamics approach to study elements of the lifecycle [51, 119]; Madachy explored an inspection-based process [105]; Lakey has also used a hybrid model for project management [93]. Raffo et al. have applied a state-based approach from systems analysis and design to the evaluation of possible process changes [137], and embed a discrete event model in a continuous framework to understand the consequences of omitting unit tests when developers are experienced [107]. Donzelli and Iazeolla propose a two-tier approach, with a discrete event queuing network at the higher level and a mix of analytical and continuous methods at the lower [45]. Scacchi [143], Drappa and Ludewig [46] and Storrle [153] have implemented a rule-based approach.

I now overview the common paradigms applied in this research along with limitations as perceived by fellow researchers.

3.3.1 System dynamics

System dynamics is an approach originally applied by Jay Forrester in 1961 to systems analysis. It was soon applied to a number of social systems, and then to software development in 1991 [1]. The main idea is that, for systems exhibiting feedback loops, unexpected results may occur as a result of feedback interactions. System variables of interest are represented as ‘levels’ and feedback loops create ‘flows’ that cause the levels to rise and fall. Feedback from individual flows is linear, and the total result for a level may be exponential increase, exponential decrease or oscillations depending upon the multiplication factors for the various flows.

The application to software development is based on the premise that software development processes “form multi-loop, non-linear feedback systems” [134] (citing Abdel-Hamid and Madnick). One such loop common in large-scale software projects is the defect injection and resolution cycle, where defects are injected at a rate dependent upon, for example, developer experience, discovered later in the process by testing and then ‘cycled back’ for resolution. The cycle re-commences as further defects are injected during resolution activity. Other elements of such a loop may be, for example, ‘schedule pressure’ and ‘productivity’ as rework causes slippage [93].

A typical modelling effort in this paradigm involves identifying a problem, developing “a dynamic hypothesis explaining the cause of the problem”, building a computer simulation model of the system and testing the model to ensure it reproduces real-world behaviours [154]. One claimed benefit of such an approach is that outputs are dependent upon all relevant factors in a ‘global way’. Lehman warns of the danger of ‘local’ process improvements and declares that “Local fine tuning cannot be expected to make a major contribution to global effectiveness” because “It is a well-known property of complex systems that local optimisation usually causes global sub-optimisation” [98]. The *System Dynamics Society* reminds us that “Only the study of the whole system as a feedback system will lead to correct results” [154].

The system dynamics approach to modelling social systems has been criticised by several researchers. Starr comments that such modelling is a form of non-experimental research and the model effectively represents a theory about the operation of a system. As hypothesis generation and testing are absent, the model is not reflective of true causal mechanisms and can not be used for predicting. He believes there is a danger of slipping into ‘prescribe’ mode when using such models [151]. Legasto and Maciariello believe models are prone to methodological problems, for example, disagreement about whether to model unmeasured variables and the suitability of the loop-cause-effect format for individual problems. Other criticisms are the lack of an empirical base and the intended use of the paradigm for long-term policy-making, both of which render the approach inappropriate for short term prediction [8]. More recent criticisms from the field of software engineering include the inability to capture attributes in system dynamics

models. For example, a model ‘level’ for a ‘code’ entity may rise and fall in size, representing, for example ‘amount of completed code’, but there is no way to capture other attributes, for example, ‘code complexity’ [107]. Because system dynamics works at a system level, limitations also include the inability to model, for example, a ‘start coding’ task while preventing, for example, ‘start design’ from commencing as soon as the ‘Code’ level became non-zero [106].

My contributions to the list of criticisms include the use of the term ‘causal loop’ when such a loop actually represents a belief about how we think software development works and the embedding of process decisions in the model, generally in a non-transparent way. The first means that there is a confusion between ‘belief’ and ‘causation’ and beliefs become unavailable for altering or fine-tuning. It also means it is not possible to model random factors (for example, an engineer is unhappy because his dog just died). The second criticism means we tightly-couple process and policy and so remove the possibility of decision-making during process execution. For example, in the feedback loop cited above, there is a loop linking ‘defects generated’, ‘schedule and effort’ and ‘staffing profile’ [93]. The embedded assumption is that an increase in rework required results in increased staffing. The ‘fixing’ of the process in this way equates to a fixing of a policy i.e. that staff numbers will be increased to handle an increased need for rework. Such a policy might be applied and then changed according to circumstances but is now embedded as an integral part of the process. Barros et. al. also comment on the mingling of facts and assumptions and suggest a strategy to effect separation [39]. I also notice that the concept of ‘feedback’ in engineering systems generally does not involve a consideration of time — it is assumed that feedback manifests quickly and results, for example, steady state, are achieved within a short time interval. When applied to social systems, the aim is generally one of understanding and again time frames are not of key interest. However, time frames are crucial in software development projects. It is important to know if the results predicted by the causal loops in the process model, for example, fewer defects, can be expected to occur within the life-time of the process! The warnings in the early literature against using the method for short term predictions [9] seem relevant to software projects. Despite all of this, the claims in the literature are that good results have been achieved when a model is calibrated with data from specific organisation.

3.3.2 Event driven

In a discrete event simulation, discrete entities (‘units of traffic’) move (‘flow’) from point to point in the system while competing for scarce resources. Entities instigate and respond to events (things that happen and change the state of the system). The system state changes at only a discrete, and possible random, set of simulated time points [145]. When this paradigm is applied to software development, the ‘product artifact’ entities flow through process blocks.

Each process accepts unique input items and creates unique output products [93]. The ‘complete activity’ event is the event that causes system change. Times for each activity are embedded in the model and sourced from a target organisation.

This approach has proven suitable for the software domain, as software processes are generally defined in discrete terms, for example, ‘design’ [93]. Efforts and durations may be based on statistical distributions, allowing the uncertainty that occurs in real software projects to be modelled [158]. In this paradigm, delays may also be captured, for example, when test equipment is unavailable.

The limitations of this paradigm relate to the fact that time changes only at event completion. This means it is difficult to model smoothly varying variables (e.g. productivity, schedule pressure) as some process blocks have a long time span. A second limitation is that the approach “restricts the software development process to a predefined sequence of activities” [46]. A direct application of this paradigm would present difficulties if we want to change the process in a non predetermined way. For example, if we want to add an inspection step, or omit the design step and move straight to coding with pair programming and test first design, we would have to change the model structure.

3.3.3 State based

As described earlier in this Section, several researchers have applied a state-based approach to modelling the software development process, an approach first suggested by Humphrey and Kellner [67]. State transitions are triggered by events relating to task commencement or completion. States describe task status, for example, ‘InDesign’ or ‘Tested’. Parallelism can be represented, for example, ‘design’ task completion might cause the system to be in a state that represents both ‘InCode’ and ‘InTestPreparation’ [158]. Wakeland points out that feedback loops cannot easily be represented because state change occurs only with events and long time frames may exist between events.

The state-based approach as described here has states that are described in terms of task status. This has limitations for flexibility, for example, a policy to ‘commence coding when designs are 80 percent complete’ is not easily captured.

3.3.4 Limitations

Simulation models are generally created for a specific company process and tend to use metrics data from the target company for model formulation. The reasons for this are that the software development process is complex with many factors influencing outcomes and there is a consequent lack of sound, comprehensive industry data on which to base models [46]. Research

aimed at an improved understanding of the keys factors that affect software project outcomes is thus difficult, and researchers focus on understanding within specific environments. This situation means that many current simulation models are essentially *products* rather than representations of a theory about how software projects work.

One consequence of this is that results from modelling studies are applicable only in the same environment. Studies generally aim at aiding specific companies to predict the results of process changes based on previous projects.

In addition to the above ‘scope-of-application’ limitations, simulation models tend to contain many assumptions relating to project contexts. Some of these assumptions manifest as integral parts of the model architecture. For example, when Abdel-Hamid and Madnick first applied a system dynamics approach to the modelling of the software process, they postulated that developer motivation decreases over long projects [1]. The form of the ‘causal’ relationship is characterised in the model equations. This relationship has appeared as an unstated assumption in almost all subsequent system dynamics models. Other assumptions are often ‘hidden’ in the metrics used to populate a model. For example, a company’s metrics database may contain a measure of ‘typical productivity’ or ‘average number of defects injected or found’, and these metrics in fact ‘hide’ the fact that real people are coding at a certain rate and with a certain level of proficiency. If we don’t know what were the human factors at play when the metrics were collected, we have no idea whether or not we may apply the same metrics in another project. In simulation experiments, the definition of what is being measured is generally clear, as the target process is generally that of a specific company. Assumptions for this approach tend to relate to contexts, as these are often either buried in the model architecture or assumed in the target data.

An interesting discussion that occurs in the simulation modelling literature relates to the capture of continuous variables. System dynamics modellers tend to claim that some variables, for example developer motivation during long projects, exhibit feedback, i.e. change throughout the process in a continuous way, and this is best represented by the system dynamics paradigm. Other modellers employ a discrete mechanism to capture such change. For example, Raffo and Harrison manage such variables by means of some persistent storage that captures changing values as the project progresses [136]. Martin and Raffo present a model in which values for continuous elements are obtained by calculating model equations at regular time intervals [106]. The use of discrete intervals to model continuously changing values is addressed in the early modelling literature [140].

Although several simulation modelling researchers have examined the theme of software process flexibility [13, 35, 46, 93, 115, 144, 153], a model that facilitates comparison of processes across modelling paradigms has not yet been suggested.

3.4 Discussion

In the Sections above, I overviewed three kinds of empirical models used to help researchers understand the software development process with a view to predicting outcomes. To my knowledge, these represent current such research. I argued that each of the three model types (predictive models, models used for controlled experimentation and simulation models) are characterised by assumptions and these assumptions impose limitations on conclusions that may be drawn from studies based on the models. One consequence of this is that some studies appear to give inconsistent results. For example, in an examination of studies involving pairs of developers creating code [84], two studies [120, 169] present results that indicate that developers working collaboratively produce better quality code with very little loss in productivity, and a third study shows that collaboration is about one quarter as productive as solo programming and concludes that it is pair programming's role-based protocol that is the cause of the good results [126]. However, the second study [120] produces good results for *collaboration* (not pair programming) and this is in direct contrast with results of the last study.

The lack of consistency in experimental results is an acknowledged problem when empirical research is not based on an underlying theoretical model [55, 88, 90]. I take this up again in Chapter 5.

3.4.1 Software measurement

I conclude this Chapter with a discussion on the problem of software measurement. According to Kitchenham, Pfleeger and Fenton, there is a problem with the integrity with which software practitioners and researchers measure software-related attributes [89]. It is important to work with validated software metrics and at the current time there is no agreed way to perform such validation. Several researchers have addressed the issue of validation. For example, Weyuker proposes a set of properties that measures must be shown to exhibit. However this set is believed by Zuse to be inconsistent. The result is that "...new measures are being justified according to disputed criteria, and some commonly-used measures may not in fact be valid according to any widely accepted criteria" [89]. This situation has not yet been resolved [91] and is problematic because "the major rationale for using metrics is to improve the software engineering decision making process from a managerial and technical perspective" [48].

Kitchenham et. al. discuss the structure of measurement and present a framework for validating measures along with a plea to the software community that discussion is needed for agreement to be reached. They believe that "...software measurement must be consistent with measurement in other disciplines" [89] and remind us that "A measured value cannot be interpreted unless we know to what entity it applies, what attribute it measures and in what unit" [89].

They also remind us that, according to measurement theory, the *scale* for a measurement is generally one of *nominal*, *ordinal*, *interval* and *ratio*, and each of these obeys specific rules for manipulation. For example, fault categories with values ‘Major’, ‘Minor’, and ‘Negligable’ might be captured using an *ordinal scale*. This means it is not appropriate to perform arithmetic manipulation on the values. A common mistake occurs when the categories are represented numerically, for example, ‘0’, ‘1’ and ‘-1’, and then combined in some way to obtain a ‘representative’ value. The problem lies in the fact that the labels are entirely arbitrary and so arithmetic manipulation is meaningless. Consideration of common software measures in the light of such errors show that many common software measures are flawed and that “a range of simple measures are valid within well-defined contexts, but also shows that certain measures cannot be deemed to be valid according to any reasonable scientific notion” [89]. The authors specifically discuss function points in this context and Kitchenham expands the discussion in [86].

There is disagreement about some of the details of the proposed framework. For example, Kitchenham et. al. believe that the unit that describes how we measure an attribute defines the scale, for example, ‘Fahrenheit’ is an *interval scale* unit of temperature, whereas ‘Kelvin’ is a *ratio scale* unit of temperature. Attributes are thus independent of the units used to measure them and “any property of an attribute that is asserted to be a *general* property but implies a specific measurement scale must also be invalid” [89]. Morasca et. al. believe such a model will present problems when there are well-understood intuitions, and cite the example of an intuitive understanding about the concept of object size, i.e. when two objects are put together, the size of the compound object is not less than the size of either constituent [113].

The fact that there is disagreement between researchers at such a basic level supports the belief of Kitchenham et. al. that the industry is working with disputed criteria and that the foundations of software metrics are very shaky. Issues of measurement are highly relevant to the problem of comparing processes because, if we cannot measure, we have no sound basis on which to base comparisons. However, for reasons of pragmatism, I regard problems of measurement as out-of-scope for this dissertation and simply work with the measures that are commonly applied by practitioners and researchers.

4

Related Work

This dissertation addresses the need to synthesise processes according to project environments in order to predict outcomes. In Section 1.4, I noted that, before we can synthesise, we must be able to represent elements of processes and process models. I also noted that, before we can predict outcomes of applying process elements, we must first be able to compare the effects on these outcomes of different elements.

In this Section, I overview a number of research areas that are relevant to the problems of representing, comparing and combining processes. In Section 4.1, I overview some existing process frameworks that provide support for project-specific specialisation, in Section 4.2, I describe some researchers' approaches to tailoring the software process to project environments and, in Section 4.3, I describe some simulation models that aim to support flexible representation of processes. In all cases, I describe what has been achieved and discuss limitations. In Sections 4.4 and 4.5, I overview some related areas of research and comment on why these are relevant to this thesis. In Section 4.4, I overview some frameworks aimed at aiding experimentation. In Section 4.5, I overview some efforts at providing a suitable abstraction for the human-related factors in a software process.

4.1 Process Frameworks

Some process models presented in Section 2.2 are in fact frameworks that allow specialisation. In this Section, I discuss these frameworks and the limitations that restrict their ability to represent any development process in a flexible way.

4.1.1 The Spiral model

The *Spiral* model was presented in Section 2.2 as a risk-driven model [23]. Spiral effectively supports flexibility in the ordering of the phases, for example, coding a prototype may take place before specifications are complete. The ‘best’ phase to action in each cycle is determined by risk assessment of objectives, alternatives and constraints. As the model is completely flexible in what are the objectives for each cycle, it is, in fact, a process framework that allows specialisation.

The model requires up-front management activity (planning, scope-setting, constraints identification and risk assessment) at the start of each cycle. Thus, although there is flexibility in the technical aspect of the model, i.e. there are no constraints on what is implemented during the cycle, the management aspect is mandated. In this sense, each cycle is like a mini-project, in that planning, scope definition and risk management are integral. Although the need for planning is acknowledged by the software industry, I observe that the planning activities included in the Spiral model represent only a subset of the activities suggested in the project management literature [135] and these activities are mandated for all software projects, regardless of size or criticality. I suggest that this represents a limitation to the model’s use as a general framework, as flexibility in the planning function is removed. I take up the issue of planning in Section 5.3.2.

I also note that, although the model has a step for identifying constraints imposed by the environment, human factors do not appear to form part of these constraints. Although one could identify, for example, ‘project manager does not communicate well’ as a constraint that spawns a risk, it would seem that this is pushing the bounds of the expected use of the model. This lack of inclusion of a ‘human factors’ aspect in fact creates an inherent risk in model usage. The assumption of risk management expertise is acknowledged by Boehm as a difficulty in applying the model, but I would also add the risks of lack of planning expertise and a lack of acknowledgement of the effects of the human-related factors.

4.1.2 The OPEN process

OPEN is an object-oriented methodology, created by the *OPEN Consortium*, that provides support for the software development process. It is based on a number of methodologies, for

example *MOSES*, *SOMA*, *Firesmith* and has an associated modelling language, *OML (Object Modeling Language)*. *Open* is described as ‘Third Generation’ because it addresses the whole process lifecycle, rather than the development lifecycle only, and provides a mechanism for tailoring of processes to suit project contexts [57].

OPEN provides a number of ‘Activities’, some relating to technical development, some to the project lifecycle and some to program planning. Examples are ‘Project initiation’, ‘Analysis and model refinement’ and ‘Project planning’. Each ‘Activity’ is realised by application of a number of ‘Tasks’ and each ‘Task’ is implemented using one or more ‘Techniques’. ‘Activities’ have pre-conditions and post-conditions, the latter resulting from the mandatory inclusion of a ‘testing Task’. *OPEN* is often referred to as a ‘contract driven lifecycle’ because an ‘Activity’ may not commence unless all pre-conditions are satisfied. Although ‘Activities’ are fixed, flexibility is achieved by selecting ‘Tasks’ and ‘Techniques’ that best fit the project environment i.e. by applying a suitable ‘process pattern’ from a number of common process patterns appropriate to different domains.

As a framework to support process flexibility, *OPEN* has a number of severe limitations. The most obvious is that the framework supports OO technology only. This limitation applies to *all* lifecycle ‘Activities’, including those concerned with requirements and planning. Although the authors make a case for the need for a seamless environment, many software practitioners disagree that OO techniques are automatically best for all software architectures [148]. In addition, the expectation that the client will always be comfortable thinking in terms of objects is quite inconsistent with the idea that analysts should speak to customers in their own language [121]. Another set of limitations relate to the failure of *OPEN* to provide comprehensive support tasks. For example, ‘Project Planning’ is included but some key planning tasks are omitted and some represented only superficially. For example, quality planning is key for many projects but there are no relevant ‘Tasks’ available in *OPEN*. The practice of *SCM (Software Configuration Management)* is also key to many projects and involves several different aspects, for example, item identification and auditing, but ‘Establish change management strategy’ is the only aspect of *SCM* represented [72]. As further examples, there is no mention of ‘Project monitoring and control’, a serious omission for projects of any size [135], and some subtasks do not obviously map to ‘parent’ ‘Activities’. For example, some subtasks for ‘Resource allocation planning’ appear to bear no relation to resource planning. A further limitation occurs in the pre-definition of ‘Activities’ and post-conditions. The latter means that every ‘Activity’ has an associated ‘testing’ or ‘evaluation’ ‘Task’ i.e. is constrained in some way and advancement is not possible until the precondition is met. The former means the process is constrained to comprise specific, predefined ‘Activities’. Although the authors suggest the *OPEN* process is suitable for small teams [57], it would appear that some agile processes are not supported. This represents a limitation when considering *OPEN* as a suitable framework for general representation of processes.

In summary, the OPEN framework permits selection of ‘Tasks’ based on project contexts but the list of ‘Tasks’ from which to select is incomplete. There is no guidance on how to select tasks. The idea is, I think, that a number of ‘process patterns’ will be developed for different contexts, presumably based on expert opinion. There is no mechanism for capturing context-related information and human aspects do not appear to be addressed.

4.1.3 The Rational Unified Process (RUP)

In Section 2.2 I described the *Rational Unified Process (RUP)*. This process is also a process framework, in that the supported process can be “adapted and extended to suit the needs of an adopting organization” [92].

RUP has four ‘Phases’, and each is implemented by a number of ‘Iterations’. Nine ‘Workflows’ are defined (for example, project management, requirements) and are active concurrently throughout all ‘Phases’ and ‘Iterations’. This provides a high degree of flexibility in process definition and both agile and traditional projects have been captured in *RUP*. The iterative nature of each ‘Phase’ also allows the possibility of risk management activity throughout the process.

As an implementation framework, *RUP* limitations include support for *OO* technology only, a lack of guidance as to what is an appropriate process for a given project and the lack of a mechanism for capturing contexts.

4.2 Process Tailoring Approaches

I now overview some approaches taken by researchers towards tailoring the software process for specific project environments. I exclude approaches based on modelling, as these are described in the next Section.

4.2.1 Basili and Rombach: Tailoring to Project Goals and Environments

In 1987, Basili and Rombach presented a methodology for “improving the software process by tailoring it to the specific project goals and environment” [13]. The methodology was aimed at improving the process within a given environment. The research involved a *NASA/SEL* collaboration. The vision presented in the Basili-Rombach paper represents one similar to that which is the subject of this thesis — a mechanism for supporting short-term data accumulation and long term process selection and tailoring.

The key idea is that improvement may be attained by identifying a productivity or quality goal for improvement, by applying the *GQM* (Goal/Question/Metric) paradigm [11] to quantify the goal, and by finally identifying the effects of selected methods and tools on the quantified

goal. The authors describe an application of this methodology in which the goal involves minimising defects (*errors, faults and failures*). The goal is quantified as the number and types of defects imposed by environmental factors and methods and tools are categorised by the number and type of defects related to their use. They suggest that other approaches to goal selection and quantification are possible, for example, involving some measure of customer satisfaction.

The authors point out that, rather than measuring the environment directly, they are “actually measuring the impact of the environment on the quality of the software process and its resulting products”. They claim that this indirect characterization has the advantage of objectivity, although this claim is supported by neither explanation nor evidence. They also point out that, for improvement to be effective, knowledge about the impact of methods and tools on defect profiles is necessary and “we do not have enough knowledge yet”. They suggest that each application of the improvement methodology will result in increased knowledge and some substitution of actual analysis results for hypotheses. An application of the approach to a ‘characteristic’ project in the *NASA/SEL* environment is presented. The process applied was a well-established one, with continuity of experienced management. The improvement methodology proved to be feasible and beneficial.

The vision for the research was to enable software development environments to include, in addition to the standard construction tools, flexibility in selecting a process model and the ability to tailor it to specific *project* goals and environments. The authors acknowledged that, before this vision could be achieved, much data had to be accumulated. They saw the methodology as being a step towards supporting such accumulation. Although the approach has been extended by the same group [110], it would appear that uptake by other groups has been minimal. Several limitations in the described approach are apparent and it is possible that these limitations have contributed towards the failure of industry to participate in the effort. I overview the limitations below.

The improvement exercise described is an experiment in which some factors (methods and tools) are perturbed to ascertain results on a quantification of a goal. The aim is that the effects of the perturbation on the goal will be better understood, providing data towards supporting tailoring. However, as a tailoring mechanism, there is a lack of holism that compromises the usefulness of the approach. For example, the goals of a software development effort generally involve more than one factor and several authors warn against the danger of focussing on a single factor [90, 93]. A more realistic expectation might be that both a certain level of functionality and an agreed defect level are achieved at some agreed cost. A second limitation relates to the identification of possible confounding factors. The authors categorise these as ‘problem factors’ (for example, the type of problem, newness to the state of the art, susceptibility to change), the ‘people factors’ (for example, number of people, their expertise), the ‘product factors’ (for example, size, deliverables, reliability and portability requirements), ‘re-

source factors' (for example, machine availability, budget) and 'process and tool factors' (for example, available tools, training, code analysers). Some of these factors really form part of the goal definition for a software development effort. For example, 'reliability requirements' are an expectation on the delivered product i.e. the product must have, in addition to certain functionality and cost, an agreed reliability performance. Although this list may be viewed as confounding factors for an experiment in which a single goal is examined, I believe this list is too unstructured to provide appropriate support for tailoring. Although the approach has proven useful for improvement within a given (and assumed constant) environment, it does not provide support for the general case.

4.2.2 Boehm and Turner: Balancing Agility and Discipline

Boehm and Turner examine the issues relating to 'agile versus traditional' process selection and believe that it is necessary to "have a repository of 'plug-compatible' process assets that can be quickly adopted, arranged, and put in place to support specific projects" [21]. They believe this can be achieved by a risk driven approach. They examine the 'home grounds' for the agile and traditional approaches i.e. the environments in which the approaches are believed to be most successful. They then specify five critical project-related factors based on these as orthogonal dimensions. Dimensions are 'criticality', 'size', 'culture', 'dynamism' and 'personnel'. Projects are charted according to their values along the five dimensions. Projects closer to the centre of the chart are 'more agile' and those closer to the edge are 'more traditional'. Standard risk management processes are then applied to select a strategy according to the perceived risks along each of the dimensions [22].

The authors state that the five project factors identified above are 'critical' and 'orthogonal', but these statements are unsupported and not discussed. It is possible that some correlation exists between, for example, 'culture' and 'dynamism'. The main limitation of this approach is that its principal purpose is to identify the kind of risks that categorise the project. Risk management techniques are required to actually choose appropriate strategies and processes. A second limitation is that it is based upon beliefs about the 'home' areas for the different process types. As pointed out in Section 2.3, the issue of what comprises traditional and agile processes is not clear cut and any categorisation is inherently approximate.

4.3 Process Simulation

In Chapter 3, I presented three research groups interested in modelling the software process in order to better understand and predict process outcomes. In this Section, I discuss some

models from one of the groups, the simulation modellers. The models discussed all in some way support flexibility in process selection.

4.3.1 Drappa and Ludewig: Interactive simulation

Drappa and Ludewig [46] describe a simulation system that allows trainee project managers to manage a simulated project interactively and view the results of decisions made on project effort and defects. A rule-based modelling language is implemented and a time-discrete mechanism used for simulation. The model is initialised with a specific process and calibrated with data from the literature, augmented with expert opinion.

In this system, a modelling language, SESAM, allows software projects to be described by a collection of rules, each of which produces a certain effect on the state of the project. Rules exist for various process granularities and are hierarchically managed, providing flexibility in the level at which users may interact with the system. Users ‘run’ a number of time steps and may issue commands to the system, for example, to assign specific developers to a task. As all rules act on a global data structure, the system may be extended by adding rules that cover different aspects of the process. Model assumptions are discussed and include, for example, decrease in developer productivity when team size increases and improved productivity and quality when developers are experienced and capable. Models may be adapted for different environments by expressing model parameters as constants rather than embedding in rules, as constant values can be easily changed.

This model allows much flexibility within certain limits. The authors report that the model is “. . . restricted to a certain class of software projects” as it proved too difficult to “. . . develop a universal model that fits any particular software project . . .”. This decision is possibly based on a potential ‘rule explosion’ (the authors report several hundred rules for a realistic model) and effectively constricts the activities provided by the model. This means that it is not possible to represent, for example, to start coding when designs are 80 percent complete. A second possible limitation is a potential mismatch between the use of ‘typical’ industry data and the scoping of the model to “. . . small to medium size software projects . . .”, as available data tends to be from large-scale projects. Other limitations relate to the model assumptions, for example, the effects of team size and developer experience on productivity and quality.

I note that abstracting the process as a number of rules may cause problems as the system grows in size. Each possible factor that affects outcomes, for example, ‘experience’, must be included in every rule that predicts outcomes and every possible combination with other factors. As the number of factors grows, the number of possible combinations will become prohibitive and so the abstraction is not scalable. In addition, the possibility of adding conflicting rules will increase. I also note that model assumptions, although acknowledged, are buried within the

model rules.

4.3.2 Lakey: Project management

Lakey [93] introduces a hybrid model to support software project estimation and management. The model is intended as a theoretical framework. It comprises a number of building block activities along with equations for each that calculate the model required values (for example, ‘number of defects generated’). Calculation inputs include values for a number of project factors that are believed to affect the results, for example, size, skill level, tool support. The internals for each block are captured as a system dynamics model in which relationships between schedule pressure, defects, etc. are embedded.

In this system, project-specific process models are built by creating an appropriate number of building blocks and calibrating the equations for each with data from the project to be modelled. Four building blocks are available — these are ‘preliminary design’, ‘detailed design’, ‘code and unit test’ and ‘subsystem integration and test’. Values for project, process and product factors are input to customise the blocks. Examples of project factors included in the model are ‘communication overhead’, ‘tool support’ and ‘skill levels’. Examples of process factors are ‘defects injected’ and ‘estimated calendar weeks’. Product factors include ‘size’ and ‘quality’.

A strength of this model is the inclusion of all of the cost, schedule and quality performance parameters in a holistic system as “the primary software project performance parameters of cost, schedule and quality are not independent, and they cannot be tracked and managed independently”. However, customisation is achieved by copying and renaming building blocks to achieve the correct process structure and then providing the relevant input values. I observe that this means that only basic building blocks as provided are available and there is no possibility of representing any tasks that do not comply with one of these blocks. I suggest that customisation thus refers to changing input values rather than changing the process. Another limitation is in the pre-definition of the factors that are believed to affect outcomes. The beliefs are effectively model assumptions.

4.3.3 Munch: Process patterns

Munch applies a patterns approach to the development of custom-tailored process models [114, 115]. He believes that “The development of high-quality software or software-intensive systems requires custom-tailored process models that fit the organizational and project goals as well as the independent contexts”. In Munch’s solution, a process pattern is a reusable fragment of a process model that represents an activity. Patterns can be combined to represent combinations of process models. Each pattern is described along with some information. This includes a

‘characterization vector’ that contains attributes such as ‘SW maintenance is false’, ‘Maximal effort is 2000’ and ‘Requirements is false’ along with a goal that describes a restriction on these attributes, for example, ‘Maximal effort is less than 2000’. Required information also includes a description of how attributes are transformed when the pattern is applied, for example, causing a change to ‘reliability’ [115]. A goal also may incorporate a ‘quality pattern’, for example, a prediction model for test effort based on design complexity, that effectively defines the transformation function.

In this model, the required goals are restrictions on project attributes. It appears that these attributes include only those over which the project has control. Transformations change attribute values, and goals are restrictions on those values. This means that the model does not include factors over which the project has no control, for example, developer characteristics or company culture, and the human element is not modelled.

Another limitation is that the transformation model implements a number of rules that apply actions (transformations) according to the value of a characterisation vector attribute [114]. For the reasons discussed in Section 4.3.1, I suggest that the rule-based nature of the abstraction will cause problems of scalability when applied to the many possible characterisation vector attributes. Munch reminds us that patterns have been applied to software design and believes that, as a reuse mechanism, their use is appropriate for the software development process. However, I suggest that the human-intensive nature of the software process renders definition of patterns prey to the same problems as definition of processes i.e. the large number of contexts that affect developer efficacy must be captured in some way. The described model appears to have no abstraction for these ‘human-related’ factors.

4.3.4 Storrie: Process patterns

Storrie [153] presents a new adaptive paradigm for software processes based on agile development ideas and suggest the use of process patterns for unifying software processes. A process pattern describes a piece of a process and is described in such a way that composition of patterns according to pre- and post-conditions is possible. Selection of patterns is subjective, however, and there is no mechanism in the model for evaluation of a resulting process against some predefined objectives.

4.4 Experimental Frameworks

In Chapter 3, I identified three groups of researchers who build models of the software development process for the purpose of greater understanding. One of the groups carries out formal experiments, and I noted that this group, although growing in size, comprises only a small num-

ber of researchers. In an effort to increase the quality of this research and empirical research in general, several authors have proposed frameworks for helping researchers plan and implement software engineering studies. Kitchenham et al. present a set of guidelines “intended to assist researchers, reviewers, and meta-analysts in designing, conducting, and evaluating empirical studies” [90]. Basili et al. [14] address the planning of experiments by using the GQM framework [11] to articulate the purpose of the study and extending this to facilitate categorising of studies. Williams et al. [170] propose a framework, XP-EF, for collecting data in XP case studies. I discuss these below and identify the contributions and limitations of each in the context of the goal of process flexibility.

4.4.1 Kitchenham et al.: preliminary guidelines

Kitchenham et al. state in 2002 that “In our view, the standard of empirical software engineering research is poor” [90]. The authors admit there are methodological difficulties applying standard statistical procedures to software experiments, but that the main problem is due to researchers with insufficient understanding of statistical techniques. The authors propose some guidelines to help improve the quality of future research efforts. They suggest that such guidelines will also increase the likelihood of combining results of related studies in meta-studies.

The guidelines suggested relate to all of ‘experimental context’, ‘experimental design’, ‘conduct of experiment and data collection’, ‘analysis’, ‘presentation of results’ and ‘interpretation of results’. They thus provide a means of achieving sound empirical results that will serve to contribute to a wider body of knowledge. Such a body of knowledge would, of course, mitigate many of the problems identified in Section 3.

The authors remind us that one goal of the guidelines for ‘experimental context’ is “to ensure that the description of the research provides enough detail for other researchers and for practitioners” and that researchers need to “identify the particular factors that might affect the generality and utility of the conclusions”. However the authors, although clearly stating the importance of recording contextual information and providing guidelines for the kinds of information to include, do acknowledge that “Unlike other disciplines, software engineering has no well-defined standards for determining what contextual information” is relevant [90]. The guidelines suggest including the target industry (for example, banking, telecommunications), the kind of development organisation, developer skills and experiences, supporting software tools used (for example, compilers, design tools) and the software processes used (for example, quality assurance and configuration management processes). This list does not include some potentially important factors, for example, ‘developer uncertainty’ or ‘motivation’ (see Section 4.5). The identification of relevant contexts remains problematic, and the authors acknowledge this and suggest research into an appropriate ontology of context.

4.4.2 Basili et. al.: families of experiments

Basili et. al. remind us that “experimentation in software engineering is necessary but difficult” [14]. As discussed previously, one problem is the large number of context variables. The authors suggest that researchers need a way to work together to obtain a cohesive understanding of experimental results. They suggest the use of a framework that facilitates replication with and without some context changes and thus deals with ‘families of studies’. Such an approach will eventually lead to “a body of evidence” that will support project management decision-making. The authors use a set of experiments with software reading techniques to illustrate their approach.

The framework proposed by the authors involves first applying the ‘Goal/Question/Metric’ (GQM) template to help categorise the experiment. The GQM approach is to identify the object of study (for example, a process or product), the purpose of the experiment (for example, evaluation, prediction, etc.), the focus i.e. the aspect of interest of the object of study (for example, product reliability, process effectiveness), the perspective (for example, researcher or developer) and the context in which the measurement takes place. The authors comment on the large number of context variables that may influence the results of applying a technique. In order to support capture of the experiment, the authors suggest classifying the object of study. For example, ‘processes’ are classified first by scope and then further categorised. The examples given are ‘Life Cycle Model’ with sub-classifications ‘Waterfall’, ‘Spiral’, etc., ‘Method’ with sub-classifications ‘Inspection’, ‘Walkthrough’, etc. and ‘Technique’ with sub-classifications ‘Reading’, ‘Testing’ etc. Experimental results are classified in a similar way. For example, ‘Effectiveness’ measures are categorised as ‘Analysis’ with sub-classifications ‘Defect Detection’, ‘Usability’, etc. or ‘Construction’ with sub-classifications ‘Reuse’, ‘Maintenance’, etc.

As for the Guidelines described in the previous Section, this framework is aimed at supporting researchers carrying out experiments, rather than providing a means of representing software development processes. However, the classifications provided by the authors describe abstractions for process and product. As described above, ‘process’ is categorised into ‘Life Cycle Model’, ‘Method’ and ‘Technique’, with sub-classification in the ‘Method’ category relating to the kind of *task* being carried out, for example, ‘Inspection’ or ‘Walkthrough’. The authors remind us that “there are many ways of classifying processes”. However, it would appear that the underlying abstraction for ‘process’ is based on the idea of a task i.e. some piece of work carried out, and there is some assumption that the name that identifies the sub-category, for example, ‘Inspection’, is well-defined.

For some efforts, this may be true, for example, if explicit instructions are available. However, I suggest that such an abstraction is fraught with danger, as mis-communication as to what exactly is done is rife within the software world. For example, I present later in this thesis a

consideration of some ‘PairProgramming’ research and discover that from three studies commonly referenced as studies about Pair Programming, two are, in fact, about collaboration and concurrency (see Section 8.2). Unless the abstraction allows us to be very specific about what is being done, assumptions will be introduced. At first glance, this appears to be a problem of software development terminology i.e. is a consequence of failing to clearly define what is meant by, for example, ‘Inspection’. However, I suggest that pre-defining a set of tasks, even if carefully specified, represents a belief about how software development should be carried out. If we are to address the issue of flexibility, we require an abstraction that allows the introduction of new kinds of task that change the product in different ways. I suggest that the framework proposed by Basili et. al. is too limiting to support the goal of flexibility.

4.4.3 Williams et. al.: XP-EF

Williams et. al. present XP-EF, “a high-level view of a measurement framework that has been used with multiple agile software development industrial case studies” [170]. The authors propose that the framework be used as a first pass at a guideline for XP case studies. The framework allows capture of the extent to which an organisation has adopted or modified XP Practices. One component of the framework relates to context factors, for example, ‘team size’ and ‘geographical dispersion’. Possible factors are organised according to categories defined by Jones [79]. Projects are first evaluated according to the five ‘critical’ factors suggested by Boehm and Turner and plotted on a polar chart (see Section 4.2.2). Anomalies are further investigated by digging “deeper into the context information”.

Limitations of the framework include application to a specific process (XP) and a ‘fixing’ of contexts for capture before the industry really knows which contexts are important and which may be ignored. The danger of such a reductionist approach is that we may regard the selected attributes as ‘truth’ rather than ‘hypothesis’ and so place more faith in results that is appropriate.

4.5 People Factors

As discussed in Section 1.2, many researchers and practitioners express the importance of the influence of human factors on productivity and quality in the software development process [3, 14, 24, 37, 34, 54, 144, 155, 157]. Early attempts to better understand these influences included consideration of both individual programmer and team. In the 1970’s and 1980’s, there was an interest in the psychology of programming and techniques from psychology were applied in the examination of individual cognitive behaviour [76]. It was observed that “programmers rarely complete one subtask before beginning the next”, but rather repeatedly alternate between understanding the problem, design, code and revision [128]. Rather

than working in a top-down way, developers work at different levels of abstraction simultaneously, alternating between levels as needed. The problem of mental models of the problem was also studied with observations that “requirements documents and client’s statement of goals are never complete”, this resulting in the “criticality of domain knowledge for interpreting requirements” [128]. Around the same time, researchers examined how team structure affected outcomes, with specific interest in the centralised *chief programmer* structure of Mills and Baker and the decentralised *egoless team* structure of Weinberg [161]. Shaw concluded that the different team structures were most suited to different conditions [33].

Despite this early interest in evidence based on theories from the social sciences and the acknowledged importance of human factors, to my knowledge, there have been few attempts to create theoretical models appropriate for the software process. Traditional software process models, for example *waterfall*, treat the process as a technological one only. Proponents of the more recent agile methodologies declare the need to value people over process, and include process elements that claim to support the developers’ ability to work well. These elements are based on the beliefs of the founders of the methodology and so represent hypotheses derived from some unstated theory of human performance. As such, the agile models provide little in the way of understanding and this contributes to the fact that questions about, for example, their effectiveness in large or distributed projects can be answered only by further exploratory studies rather than by forming hypotheses based on an underlying theory.

The field of software engineering is not alone in being slow to look to other disciplines for knowledge that might improve performance. Douglas Stewart informs us that the field of operations management asks “why, when there are so many opportunities, have we paid so little attention to psychology in our research?” [152]. He suggests that the reason might be that the field has historically studied manufacturing processes in which human inputs are minimal. He also suggests that the human aspect is now more important because human-centric processes are now being studied, and the role of humans has been elevated in many manufacturing operations.

There is little disagreement that the human element is key for the software process. Many would agree with John Finan, recent winner of the Motorola ‘future vision’ scholarship competition, who believes “A great technology is one that uses the human brain as a core component” [78]. I overview three research efforts that involve creating models of the human element. The first study presents suggestions for a model based on observation and the second two based on theory from other disciplines.

4.5.1 Layered behavioural model

Curtis, Krasner, Shen and Iscoe, in 1987 and 1988, reported the results of a field study of large software development projects [37]. The authors remind us that, if a model of the software

development process is to help increase productivity and quality, it must accurately reflect what happens i.e. must “represent the processes that control the largest share of the variability in software development”. They believe that models such as the *waterfall* model reflect a *management* orientation and provide no insight into the actual development processes. They also submit that Osterweil’s proposal that software engineering processes should be viewed as software products [122] is flawed because of the variability of the processes being specified. This is a result of skill differences in the developers, degree of exposure to customers and other factors.

The purpose of the field study was to determine what are the high-leverage factors through empirical research and to describe how these factors exert their influence during the design process. The authors believe that for larger projects key factors are more likely to relate to project-and organisational-level factors, as these will tend to swamp the effects of cognitive and motivational effects of developers. They use a layered model for organising observations. The innermost layer represents the *individual* developer, the next layer, the *team*, the third represents the *project* and then the *company* and *business milieu*. They suggest that when they “overlay these behavioral processes” on the traditional technological ones, they gain insights into inefficiencies in the process.

The field study involved nineteen projects ranging in size, application domain and key system characteristics, for example, real-time, embedded, etc. The authors expected, and found, that, for small projects, individual factors would exert greatest influence on outcomes and, for very large projects, organisational factors would have most weight. Many interesting observations were made. There was a tendency for coalitions to form i.e. where a “small subset of the design team with superior application domain knowledge often exerts a large impact on the design”. There was also a tendency for developers to spend some substantial time ‘rediscovering’ existing knowledge. For example, customers might generate operational scenarios while determining requirements, but these were not recorded. Developers then tended to be unable to envisage problematic conditions.

The three main problems exposed by Curtis et. al. [36] are the thin spread of application domain knowledge, fluctuating and conflicting requirements and communication and coordination breakdown. The conclusions from the study are that “developing large software systems must be treated, at least in part, as a learning, communication and negotiation process” [34] and that developer uncertainty resulting from the above problems plays a key role. The implication is that any descriptive model of the software development process must abstract the factors that represent these problems and overlay the abstraction on the technological one.

4.5.2 Human competencies model

Acuna and Juristo reference research in human resources management and psychology and propose a mechanism for assigning people to software development roles according to behavioural competencies [3]. Their solution is based on the Logic of Core Competencies, a logic that is “practiced in many organisations for different purposes, such as personnel selection and recruitment . . .”. They see their model as a first attempt to develop the logic of core competencies for the software process.

A key element in the model is the ‘capability’ or ‘behavioural competency’. These include intrapersonal skills, such as ‘Independence’ and ‘Tenacity’, organisational skills, such as ‘Environmental orientation’ and ‘Discipline’, interpersonal skills, such as ‘Empathy’ and ‘Sociability’ and management skills, such as ‘Group leadership’ and ‘Planning’. Capabilities are then tabled with personality factors from a standard psychometric test to obtain a ‘capability-person’ relationship and with software development processes to obtain a ‘capability-role’ relationship. For example, a personality factor ‘Dominance’ is mapped to capabilities ‘Independence’ and ‘Group leadership’ and the role ‘Designer’ to a number of capabilities, including ‘Analysis’ and ‘Decision making’. Capabilities of people and roles are then matched to achieve a ‘best’ assignment of people to roles. For example, a developer with a personality profile that suggests a high ‘Empathy’ capability may be preferred over a ‘low empathy’ colleague for a customer-related role where ‘empathy’ is included in the capability-role profile.

This work is important as it is an attempt to abstract human factors in a way that is based on theory, rather than on ad-hoc and undefined values, such as ‘experience’. It also represents an early attempt to ‘match’ people with tasks based on a theoretical model.

4.5.3 Team behaviour model

Acuna, Gomez and Juristo reference research in social psychology to propose a model for team performance in the software domain [4]. They then apply the model to agile and heavy-weight development strategies with a view to finding heuristics for effective team forming. The model includes people-specific, task-related and team behaviour components. People-specific components include aspects of personality (for example, ‘Extraversion’), knowledge, skills and abilities and preferences (for example, ‘Innovative’ or ‘Conservative’). Task components include factors such as ‘Routine’ or ‘Creative’. Aspects of team behaviour include, for example, ‘Team vision’.

This work also represents an attempt to use models found to be effective in other disciplines to abstract people and tasks and match people and tasks in an appropriate way.

5

Case for a Theoretical Model

In this Chapter, I discuss some perspectives on modelling and show that, if we are to predict in a general way, models must be based on an underlying theory. I next overview some perspectives on research and note that, if we are to establish cause-and-effect relationships, research must be based on a theoretical framework.

I then apply these ideas to current software process research. I conclude that research models are not based on theory and so cannot be used to establish cause-and-effect relationships or predict in a general way. I propose the need for a theory-based model of software development and provide quotes from a number of researchers to support this proposition. I establish some objectives for such a model and present a discussion on an appropriate scope for a candidate model. I finally present my approach to the presentation of evidence to support a candidate model's ability to meet the stated objectives.

5.1 Research Perspectives

5.1.1 Modelling basics

In Chapter 3, I showed that existing models of the software process contain many assumptions. In order to clarify what is the basis of these assumptions, I now discuss why researchers build models and provide a general overview of the characteristics of different kinds of models.

According to Seidewitz, models may be created to specify a system to be built or to describe an existing system [146]. An example of a model for specification is a *UML (Unified Modelling Language)* model that describes how an OO software system will be constructed. Models that describe existing systems include, for example, Newton's model describing celestial motion and the models presented in Chapter 3 that describe the software process.

Models that describe existing systems capture relationships between various system components. It is possible to describe relationships without understanding why the relationship exists. An example supplied by Kitchenham et. al. concerns the documenting of a relationship between 'cyclomatic complexity number' (*CCN*) and 'number of faults' [90]. Although some correlation is observed, the reasons for the correlation are unknown. Another example is given by Kepler's three laws of planetary motion. As described in Section 1.3, these were based on existing planetary data and, although they appeared to successfully predict the motions of the planets, no-one knew why the laws worked. The laws describe correlations rather than cause-and-effect relationships.

Models that describe correlations should be based on data or observations. One main characteristic of such models is that they can be used to predict only in circumstances that exactly match those in which the observations were made or data collected. Kepler's laws do not apply to other celestial bodies or planetary systems. Relationships between cyclomatic complexity and numbers of defects are not guaranteed to be the same for different pieces of code produced in different circumstances. The main reason is that, for correlations, we do not know what is the real cause of the relationship and so have no way of knowing if the causal factors remain the same in the new circumstances. If a model that describes a correlation is used in a circumstance other than the one in which the observation was made or data collected, it embodies an implicit assumption that other factors do not matter i.e. an assumption that the model represents a causal rather than correlative relationship.

For a model to be used to predict in a general way, the model must be based on cause-and-effect relationships i.e. on an understanding of what are the causal factors. Newton's Laws are an example of a causal model. As described in Section 1.3, Newton postulated a force between all objects with mass in the universe and this understanding enabled him to predict accurately for all celestial bodies. The key characteristic of such models is that they are based on some theory or theoretical framework. Should such a model fail to predict accurately, it is understood that the underlying theory is incorrect. An example of this is the discovery that Newton's Laws explain universal forces for bodies moving at speeds much slower than the speed of light, but break down for high-speed particles such as are found inside atomic nuclei. Although Newton's Laws do not correctly explain *all* motion, they continue to describe a causal relationship within a well-understood domain i.e. that of constant space and time.

One way to distinguish between correlative and causal relationships is to establish if the

relationship describes an alteration over space and time. For example, a mark on a ball moving through the air will change as the ball moves [165]. A model based on cause-and-effect relationships describes a *theory*. Such a model may be used in a predictive way and may be proven incorrect should predictions prove false.

In summary, models that describe systems are based on relationships that may be correlative or cause-and-effect. Only cause-and-effect relationships support general prediction. Models based on such relationships represent theories.

5.1.2 Research basics

In Section 1.2, I commented on the scarcity, fragmentation and varying quality of existing software process data and suggested that this is in part a consequence of the immaturity of the field and its approach to research. In an attempt to place research efforts in perspective, I now present a brief overview from the field of psychology of the various modes of research data collection. I show that the only way to establish cause-and-effect relationships is to create and test hypotheses based on theoretical models.

In 1990, Gilmore described four modes of research data collection. These are *Hypothesis testing*, *Comparisons*, *Evaluations* and *Explorations* [55].

The aim of hypothesis-testing is to determine a causal relationship between two factors of interest. In this paradigm, a single factor (the independent factor) is manipulated for the purpose of discovering if the manipulation causes change to the second (dependent) factor. Everything in the experimental situation other than the essential manipulation is held constant. An example given concerns a theory that claims that code “comprehension is attained through an initial analysis of syntactic structure and, therefore that the use of indentation to indicate syntactic structure will lead to improvements in all aspects of program comprehension” [55]. A simple hypothesis-testing experiment might involve supplying subjects with indented or non-indented programs and asking them to perform tasks that may, or may not, require comprehension of the syntactic structure. Gilmore reminds us, however, that line wrapping, for example, may render the experiment impure as now any observed effect may be affected by indentation or line-wrapping. The researcher must now introduce ‘unrealistic’ conditions, for example comparing with and without line wrap, in order that the hypothesised effect might be properly observed.

According to Gilmore, a theoretical framework is vital to this kind of research [55]. Kitchenham et. al. also believe that hypotheses must be based on an underlying theory. They provide an example relating to a relationship between CCN and ‘number of faults’ and suggest that better understanding would be achieved if cognition and problem-solving theories were applied to establish the causal nature of the relationship [90].

Because the aim of this kind of research is to establish causation, I notice that researchers

must be sure that the observed effect is due to the hypothesised cause i.e. that the experiment has *internal validity*. I also notice that, as a result of the causal nature of the observed relationships, issues of *external validity* are not relevant.

Comparisons are similar to hypothesis testing but are intended to observe rather than explain any effect. Here, the researcher attempts to discover which of a number of alternatives is 'better' according to some metric. In this paradigm, the establishment of relationships often gives rise to questions or ideas about possible causality. This approach is "excellent at stimulating hypotheses and theoretical frameworks" [55]. In the example above, the research question might be whether indentation provides a more useable representation of a computer program. Line wrap is now not an issue, as it is a necessary feature of the indentation of real programs. Although results of such comparisons are traditionally tested for statistical significance, a more useful measure would be the *effect size* i.e. the size of the difference between the two conditions [55]. The CCN example above may be viewed as a Comparison as the investigation concerns an observation that high code complexity is often correlated with high fault numbers.

I note that, in this kind of research, issues of *external validity* are key as the lack of consideration of confounding factors generally renders results inapplicable in other circumstances.

Evaluations are similar to comparisons but tend to occur when we are asking a question, for example, "Can people use flowcharts?", rather than "Are flowcharts better than structure diagrams?". The intent may be to improve some weakness in a system. In this paradigm, many measures may be used, for example, subjective, human preferences may be as important as performance. Decisions about which measures are most useful may be made post-hoc [55].

Exploration involves collecting data to answer a question of the kind "What happens if ...?". This approach is most often used when new paradigms are being studied i.e. when there is insufficient data for other kinds of data collection. This data can be very difficult to collect and record and is usually not well-defined. Analysis can be time-consuming [55]. Such studies are often used as a first step and the aim is often to better understand what might be important factors in a system by capturing what is observed in a relatively unrestricted way. Results from such studies often form the basis for further evaluative, comparative or causal investigation.

In summary, research data is collected for a number of different reasons. The most general is exploration. Results from explorations often form the basis of evaluations and comparisons. Results from all may provide insights into possible causal relationships and these are investigated via hypothesis testing. This paradigm must be based on some theory or theoretical framework.

5.2 Case for a Theoretical Model

5.2.1 Need for a model

Above, I provided examples from fields other than software engineering to show that, if we are to predict in a general way, we must create models based on causal relationships and, if we are to establish causal relationships, we must have theoretical models.

The above discussion, when applied to the models described in Chapter 3, exposes the fact that these models are not based on theory and are thus correlative in nature. Simulation models are based on observation, usually within a specific circumstance. Predictive models are based on data from previous projects. Controlled experiments are based on hypotheses, but these are not theory based. One definition of the term ‘hypothesis’ is “A hypothesis is a suggested explanation of a phenomenon or reasoned proposal suggesting a possible correlation between multiple phenomena” [166]. No causation is implied in this definition. However, according to Gilmore and Kitchenham, causation is key if general predictions are to be supported. An alternative, more appropriate, definition of ‘hypothesis’ is “A tentative explanation for an observation, phenomenon, or scientific problem that can be tested by further investigation” [65].

The above observations suggest the current situation for software process modelling is analogous to the situations from history presented by Rivett [139] (see Section 1.3). Research into the software process is generally carried out without reference to a theoretical framework and is, by Gilmore’s definition, comparative, evaluative or explorative in nature. Predictive models are based on either observations or data and are correlative in nature and so incomplete. Rivett believes that, for true predictive ability, a model must be holistic and based on understanding of the system being modelled. The correlative nature of existing software process models means that the models cannot be used for predicting in a general way.

5.2.2 Support for a model

In the previous Section, I proposed that current research does not support exposure of cause-and-effect relationships and that current model-building exhibits characteristics of lack of holism and prediction without understanding. I complete this Section by presenting comments from a number of researchers who also propose the need for a theoretical framework to support prediction and research.

When discussing context for formal experiments, Kitchenham et. al. state the need to be wary of oversimplification, as in the real world, techniques are carried out within rich industrial settings [90]. An example given is a study that shows the failure of inspection meetings to decrease defects. The authors believe such a study is too simplistic because other possible benefits of inspection meetings may be missed, for example, promotion of teamwork, technology

transfer, identification of root cause detection and increased conformance to standards. One possible result of such oversimplification is that a positive result might be taken as representing the situation and used as a basis for change, when in reality the result represents a ‘local maximum’ only. For example, a technique that is shown to improve the identification of defects (local maximum) might be implemented despite the fact that it also results in higher cost, higher risk and increase in developer frustration. The advantages may not outweigh the disadvantages, and the implementation thus represents a sub-optimisation of the whole system. I suggest that problems of oversimplification and sub-optimisation are a direct result of studying parts of a system in isolation without understanding of causal relationships.

Gilmore reminds us that hypothesis-testing is the only paradigm suitable for uncovering causality and that “Hypothesis-testing research is not possible without a theoretical framework ...” [55].

Basili et. al., when proposing a framework for experimentation, declare that what is required is “... a set of unifying principles that allows results to be combined ...” and that “The ultimate objective is to build up a unifying theory ...” [14].

Fughetta believes the field lacks foundations “...related to a better understanding of the activities that constitute the software development process ...” [54]. He also believes we must “increase the emphasis on problem analysis” and “pursue long-term research goals” [54].

In Section 2.1, I state that Kitchenham and Carn view the software process as an engineering discipline, albeit an immature one. They state that “...before software engineering can mature as an engineering discipline, practitioners need a better understanding of the process by which software is created” and of the risks associated with the process [88].

As mentioned in the last Section, Kitchenham et. al. believe that a problem occurs when researchers state hypotheses that are not based on any underlying theory. For example, documenting a relationship between ‘cyclomatic number’ and ‘number of faults’ does little to expand industry knowledge as no causal mechanism is known. We can better understand the relationship in question if cognition and problem-solving theories were applied. “Without any underlying theories, we cannot understand the reason why empirical studies are inconsistent” and “Without the link from theory to hypothesis, empirical results cannot contribute to a wider body of knowledge” [90].

Dawson et. al. believe that the discipline of software engineering “needs to move towards being a rigorous discipline” and that “...theories and hypotheses have to be formed...” and “...new ideas must be advanced...” [38].

In Section 1.3, Rivett states that a model may be predictive without being explanatory, but an holistic, explanatory model is always predictive.

Scacchi believes that “contemporary models of software development must account for the interrelationships between software products and production processes, as well as for the roles

played by tools, people and their workplaces” [144].

Basili et. al. remind us that, when carrying out controlled experiments, “. . . it’s hard to know how to abstract important knowledge without a framework for relating the studies” [14].

A theory is a model or framework for understanding. Building theoretical models represents an accepted approach to predicting and generating hypotheses for better understanding cause-and-effect relationships. My thesis is realised as a theoretical model of the software development process. This represents a step towards establishing cause-and-effect relationships and predicting process outcomes.

5.3 Approach

I have made a case for a model of the software development process that provides a theoretical framework for prediction and research. As discussed earlier, the conventional scientific approach is to spawn hypotheses based on the theory or model of interest and carry out formal experiments that aim to disprove the theory.

At this time, it is not clear what will be the form of a theoretical model for the software development process. In order to progress down the path of defining a candidate model, I consider what such a model must be capable of i.e. what are the kinds of things we should be able to do with the model. This approach is equivalent to that of designing a system according to a number of objectives. If I can create a model that describes the software development process in a way that satisfies the stated objectives, I can propose this model as a representation of a theory of the software development process. At this point, the model is available for testing in the usual way, by carrying out formal experiments based on the model.

5.3.1 Objectives

My thesis is that it is possible to represent software development processes and process models in a way that allows us to compare processes and process models for the purpose of constructing new processes. I propose that model objectives are the ability to:

- Capture any software process or process model.
- Compare processes and process models.
- Create a new process by combining elements from different processes.

My approach is to aim to create a solution that satisfies the above objectives. The three objectives stated are very broad. For example, there are many aspects of processes, including

granularity and formality. The objectives must now be expanded to be more specific. In Figure 5.1, I show the three top level objectives of ‘Capture’, ‘Compare’ and ‘Combine’ and have expanded the ‘Compare’ objective to include some different kinds of required comparisons. This Figure represents a top-level specification for a suitable model.

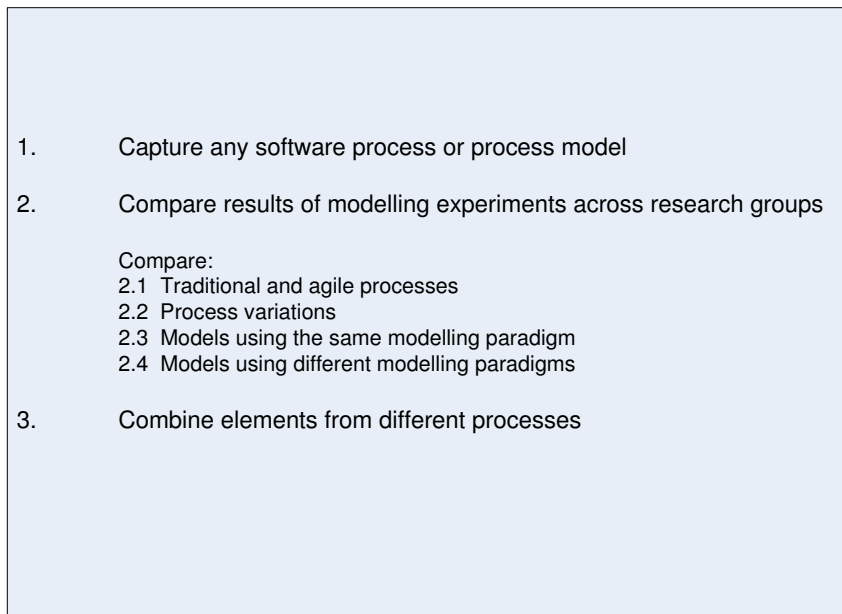
- 
1. Capture any software process or process model
 2. Compare results of modelling experiments across research groups
 - Compare:
 - 2.1 Traditional and agile processes
 - 2.2 Process variations
 - 2.3 Models using the same modelling paradigm
 - 2.4 Models using different modelling paradigms
 3. Combine elements from different processes

Figure 5.1: Model objectives

In a similar way, I expand the ‘capture any software process or process model’ objective in Figure 5.2. The first level expansion includes clauses for ‘development processes’, ‘support processes’, ‘product-line processes’ and ‘miscellaneous processes’. There are many different aspects of ‘development processes’, all of which must be addressed by a suitable solution. The next level expansion thus includes aspects of granularity, process participants (students or professionals), project size, maturity levels, etc.

The form of the objectives is similar to that of a specification for a software system. This is a useful model for stating objectives in a complete and unambiguous way. The specification presented in Figures 5.1 and 5.2 does not obey the rules for sound requirements specifications [74]. For example, it is incomplete and the meanings of some terms are not defined. Some terms are relative, for example, ‘Very large’, and there is potential overlap between objectives. An ‘Other’ category is probably required in each group. However, I believe it forms a useful starting point for model creation and the meaning of its clauses is sufficiently clear for the purpose of model definition.

<p>1. Capture any software process or process model</p> <p>Capture:</p> <p>1.1 Software support processes and process models</p> <p>1.2 Software development processes and process models</p> <p>1.2.1 Process paradigms</p> <p>1.2.1.1 Traditional</p> <p>1.2.1.2 Agile</p> <p>1.2.1.3 Open source</p> <p>1.2.1.4 Open</p> <p>1.2.2 Research studies</p> <p>1.2.2.1 Simulation models</p> <p>1.2.2.2 Predictive models</p> <p>1.2.2.3 Controlled experiments</p> <p>1.2.2.4 Quantitative studies</p> <p>1.2.2.5 Qualitative studies</p> <p>1.2.3 Process granularity</p> <p>1.2.3.1 Large-grained</p> <p>1.2.3.2 Medium-grained</p> <p>1.2.3.3 Small-grained</p> <p>1.2.4 Process variations</p> <p>1.2.5 Kinds of participants</p> <p>1.2.5.1 Industry</p> <p>1.2.5.2 Students</p> <p>1.2.6 Project size</p> <p>1.2.6.1 Very large</p> <p>1.2.6.2 Large</p> <p>1.2.6.3 Medium</p> <p>1.2.6.4 Small</p> <p>1.2.6.5 Tiny</p> <p>1.2.7 CMM levels</p> <p>1.2.7.1 CMM level 1</p> <p>1.2.7.2 CMM level 2 or 3</p> <p>1.2.7.3 CMM level 4 or 5</p>	<p>1. Capture any software process or process model</p> <p>1.2.8 Organisational paradigms</p> <p>1.2.8.1 Outsourcing</p> <p>1.2.8.2 Co-located projects</p> <p>1.2.8.3 Distributed projects</p> <p>1.2.9 Product stage</p> <p>1.2.9.1 New development project</p> <p>1.2.9.2 Upgrade project</p> <p>1.2.9.3 Maintenance</p> <p>1.2.10 Project objectives</p> <p>1.2.10.1 Standard (cost, quality, content)</p> <p>1.2.10.2 Non-standard goals (e.g. business value)</p> <p>1.2.10.3 Developer-oriented</p> <p>1.2.11 Product types</p> <p>1.2.11.1 Data-intensive</p> <p>1.2.11.2 Web</p> <p>1.2.11.3 Real-time</p> <p>1.2.11.4 Embedded</p> <p>1.3 Product-line processes and process models</p> <p>1.4 Miscellaneous processes</p> <p>1.4.1 Developers have a discussion</p> <p>1.4.2 Coding standards</p> <p>1.4.3 Add developers late to a project</p> <p>1.4.4 Developers get more enjoyment doing XP</p> <p>1.4.5 Parallel tasks</p> <p>1.4.6 Open source milestone release</p> <p>1.4.7 Project retrospective</p> <p>1.4.8 Technology transfer</p>
---	---

Figure 5.2: Objectives for 'Represent'

5.3.2 Scope

Before continuing, I discuss what constitutes a 'software process'. For some time now, I have been concerned about the tendency within the discipline of software engineering to tightly-couple different aspects of software product creation. For example, there is a trend towards 'integrated' solutions that provide support for both development and project management activities, as exhibited by the Spiral and RUP models and the OPEN process. There is also a tendency to create an entire process solution tied to a product creation technology. For example, the OPEN process mandates an OO approach for all process and project activities.

I believe that software product creation and software project management are different kinds of functions that should be represented by different processes. I provide support for this belief by discussing content from the *Project Management Body of Knowledge (PMBOK)* [135]. PMBOK is produced by the *Project Management Institute* and describes the knowledge and practices that are generally accepted by project management professionals.

According to the *PMBOK Guide*, the key project management activities are partitioned into

the *Knowledge areas* of *Project Integration, Scope, Time, Cost, Quality, Human Resource, Communications, Risk and Procurement Management*. The intent is that *all* projects require application of these processes i.e. these are the accepted and commonly practised project processes. It does not matter what kind of product or service is being created by the project. Different application areas may add some new processes that are specific to that area. For example, in the construction area, there are special practices relating to procurement and in bioscience to regulatory requirements. The suggestion is that these specific product-related processes may be included *in addition to* the core processes (and not *substituted for* any of the core processes) [135]. The PMBOK Guide thus suggests that life cycle processes specific to the kind of product being produced, in this case, software processes, should include only those activities not already represented in the core management processes. The issue is one of re-inventing the project management wheel.

I suggest that the tendency for existing process models to include aspects of project management processes is counter to the recommendations made in PMBOK. I also suggest that a lack of understanding of what are project management processes has been the source of some disagreements between process advocates.

A first example concerns the misunderstanding of the role of ‘scope management’. Scope management is a project management process for controlling changes to project scope. Project scope involves capturing justification for the project, identifying the intended deliverables, for example “. . . the major deliverables for a software development project might include working computer code, a user manual, and an interactive tutorial . . .”, and controlling scope changes, for example, a request to omit the interactive tutorial from the set of deliverables. Scope management thus includes the identification of all the deliverables to project stakeholders. This has nothing to do with requirements management, which is a product-related activity and concerns describing what the product will do. A project delivers to a number of different stakeholders. In addition to delivering a product to some end customer, it may be required to deliver, for example, progress reports to managers or designs and test rigs to the sponsoring organisation. For a software project, the ‘organisation stakeholder’ may require delivery of requirements, designs and test beds. I submit that the decision whether or not to create design artifacts is not a technical one but is rather a scope planning one. This viewpoint marries well with the need to consider product-line planning, where a single project is just one step in a potentially long chain, and technical members of the project team have insufficient visibility to be able to make appropriate decisions about what documentation should be delivered.

A second example involves risk management processes. Risk management is concerned with identifying situations that might cause a project to fail to meet its objectives, defining strategies to minimise the occurrence and impact of such situations and then monitoring for situations and effecting strategies as appropriate. This raises two points for clarification with

respect to software development processes. The first is that the identification and monitoring processes continue throughout the project. It is project management's job to select suitable point of visibility into the development process and, at these points, check for the occurrence of any risk situations, react accordingly and repeat the identification and strategy definition in the context of the current situation. This is standard risk management practice. There is no need to include risk practices in software processes. The second point for clarification is that, for software development, a particular process might be indicated as a result of such identification. For example, a mitigation strategy for requirements uncertainty might be 'have a customer representative on site' and one for product criticality might be 'complete formal reviews for all components'. Different process solutions will be appropriate for different risk situations. The point here is that it makes no sense to claim of a process that it "addresses risk at all levels of the development process" [15]. Any process will be an effective mitigation strategy for some risk situations but such mitigation is required only if the situation eventuates. If everyone knows what is the product to be created, there is no need for an on-site customer. The key idea is that such situations will be identified by the project management function.

For the above reasons, my intended model will be scoped to the *product*-related software process and will not aim to include any project management processes. This means the requirement to 'capture any software process or process model' will fail to be met for those processes and process models that comprise both kinds of process. Examples are Boehm's spiral model, in which a risk management step is mandated in each cycle, and the OPEN process, which includes some project management activities. My solution will aim to include only the product-related processes for these examples. I would argue that, in a 'bigger picture' model, the software process would be expected to neatly 'slot into' the project management ones in the same way as a 'construction process' would. The spiral and OPEN models would fail to do so and violate the 'add-not-replace' expectation stated in the Guide [135].

5.4 Evidence Strategy

In order to judge a candidate solution model, I must show that the model satisfies the objectives recorded in Figures 5.1 and 5.2.

Sources for an appropriate approach are the fields of psychology and social science, and the subset of the software engineering community interested in matters of evidence [38, 87, 160]. These sources suggest that, for people-intensive systems, an appropriate approach is to accumulate an 'evidence portfolio' i.e. a varied accumulation of evidence that helps to support the target theory or model. Such evidence might be gathered by means of formal experimentation, case studies, expert evidence and other techniques. The main considerations are that the breadth

of evidence and the degree to which we may trust the evidence are transparent.

My strategy will be to accumulate such a portfolio for the stated model objectives. In this thesis, I take a pragmatic, risk-based approach to evidence accumulation and aim for breadth and coverage rather than depth in the first instance. The reason is that the attempt to create an appropriate abstraction of the software process is exploratory in nature i.e. it is not clear that such an abstraction is possible. The approach is thus to try to accumulate evidence to support different kinds of objectives and to include objectives that appear the most difficult to satisfy.

6

Model Properties

In the previous Chapter, I presented a case for a theoretical model of the software development process and captured a set of objectives that should be satisfied by any candidate model.

As a prelude to constructing a candidate model that meets the objectives presented in Section 5.3.1, I consider what might be some of the model properties. The aim is to establish a set of criteria against which a candidate model might be judged prior to formal evaluation in order that some confidence be gained that the candidate is likely to be successful in meeting the stated objectives.

In order to establish properties, I first identify what are some of the characteristics of existing processes that must be represented. I then identify what are the limitations of existing predictive models that render them inappropriate for general process representation. I then consider some situations from real life projects and identify some characteristics that need to be addressed. Finally, I extract a number of properties required for a candidate model.

6.1 Properties Source

6.1.1 Process characteristics

From the discussion in Section 2.2, I observe the following:

1. A *task* is defined informally as “A piece of work carried out by one or more engineers” (see Appendix A). In many cases a single task name is used to describe tasks that differ in their effects on the product. For example, the term ‘design’ is used to mean many things, including ‘create formal designs from formal requirements’, ‘create formal designs from discussions with the customer’, ‘create formal designs, review the results and correct any defects’, ‘create code using informal design strategies’, etc. This means it is difficult to know when we can compare tasks.
2. There is great variation in the kinds of tasks that are carried out. For example, how do we capture the effects of ‘discuss requirements with the customer’ or ‘team meeting’? These tasks do not change the product but many authors agree they have an important effect on project outcomes.
3. The term ‘process’ is used to cover anything from a complete lifecycle, for example, ‘waterfall’, to a single, small task, for example, ‘review designs’. Processes are described at different levels of granularity and this means it is difficult to know how to compare and construct processes.
4. Many processes are characterised by tasks being carried out in parallel, for example, designs and test plan production. When the same area of the product is affected, for example, developers working on the same piece of code, there are potential problems with defining what is the correct version.
5. There is inconsistency in what aspects of the product are measured and no mechanism for categorising or comparing measurements. For example, can we compare tasks if one results in change to ‘Lines of code’, another to ‘Number of requirements implemented’ and a third to ‘Number of stories implemented’?
6. There is interest in representing different kinds of product-related objectives, for example, those relating to economic value. For example, some authors believe that it is necessary to capture the business value of a product and this must be done throughout the project to reflect value-related attributes at each stage in the process.
7. Processes include different assumptions about how human factors affect outcomes. For example, traditional processes assume issues of team size are the only relevant ones but some authors believe many other factors affect outcomes.
8. Some processes claim that the developers involved change as a result of participation. For example, it is claimed that developers become more confident and are more satisfied as a result of participation in an XP project.

9. Some process descriptions mandate a particular technology, for example, OO. Such descriptions can not be used to represent processes in a general way.
10. Many believe it is not possible to represent processes in a deterministic way. For example, modellers often handle what is regarded as inherent uncertainty by taking inputs from statistical distributions and presenting study results as means and standard deviations.

6.1.2 Model limitations

In Chapter 3, I identified the following limitations in current research models:

1. Research models tend to report only some objectives and so fail to give the whole picture as regards model efficacy. For example, many studies report data on numbers of defects but do not provide associated cost data. The disadvantages of this were discussed in Section 5.2.2.
2. Models embed different beliefs about how human characteristics affect outcomes. For example, many system dynamics models include ‘developer motivation’ and the assumption that this decreases over long projects.
3. Many measures are applied without a clear statement of what these measures mean and manipulations on these measures is often inappropriate (see Section 3.4.1).

6.1.3 Real-world situations

I now consider some real-world situations that should be addressed by a candidate model. First I discuss product lines. In Section 5.3.2, I pointed out that the deliverables from many projects comprise part of a product line. In this situation, a single conceptual product exists separate from an individual project. I illustrate this situation in Figure 6.1. Product ‘MyWebApp’ is transformed by three different processes, ‘A’, ‘B’ and ‘C’. Two of these occur in parallel i.e. ‘MyWebApp’ is in two different states at the same time.

This diagram also illustrates another ‘real-life’ scenario. It is possible that an industry project is required to deliver its product in more than one state. For example, a project tasked to deliver version 1.0 of a new application might be expected to also make a pre-release delivery to an ‘early-adopter’ customer.

In Section 5.3.2, I noted that the organisation that sponsors a project is a stakeholder and may require that the project delivers documentation to it, for example, architectures and designs. This is likely in the case of product line projects but can apply to any project. This means that

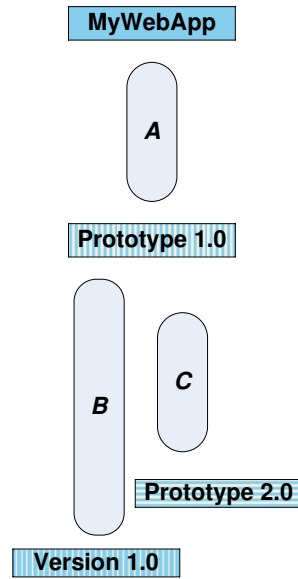


Figure 6.1: Product and process

all artifacts delivered to all stakeholders should be considered as being part of the delivered product.

In Section 5.3.2, I also discussed the existence of the *Project Management Body of Knowledge (PMBOK)* [135] and stated the need for a clean interface between the processes used to create the product (the *software* processes) and those used to manage the project (the *project management* processes). The result of the discussion is a decision that a candidate model should represent software processes only.

Finally, I address the issue of ‘readiness for delivery’. Projects commonly deliver according to some agreed ‘quality attributes’. For example, it may have been agreed in a Quality Plan that delivery may take place when the only known defects remaining after testing are unlikely to cause incorrect product functioning.

6.2 Properties

From the above, I extract the following properties.

P1 Only processes that directly affect the software product are represented (described in this dissertation as *software development processes*). In particular, project management processes as defined in *PMBOK* are not included.

P2 Product represents all descriptions of all artifacts that are delivered to all stakeholders. This includes problem descriptions, for example, requirements, and solution descriptions, for

example, designs.

- P3** Product may be represented by a number of different measures. For example, representation might be ‘lines of code’ or ‘number of requirements’.
- P4** Product may be represented by more than one measure. For example, representation might include all of ‘number of requirements’, ‘number of defects’ and ‘number of person hours’. This allows representation of, for example, both quality- and cost-related attributes.
- P5** Product representation should be extensible in that new attributes can be included.
- P6** Processes may be represented at any level of granularity. For example, ‘create product’ or ‘carry out code inspection’.
- P7** Task definition is unambiguous. For example, for a task ‘design’, it is clear what the task changes and how it performs the change.
- P8** A task may result in change to the humans carrying out the task and some tasks result in change to humans only. For example, developers become more satisfied as a result of participation in an XP project and design discussions do not change the product.
- P9** Different beliefs about how human factors affect project outcomes may be represented.
- P10** Some notion of ‘readiness for delivery’ is represented and its use optional.
- P11** The model should account for product line processes, where a single conceptual product is changed by several projects and projects often deliver a product in more than one state.
- P12** Task parallelism should be supported.
- P13** The model should be technology-independent.
- P14** The model should represent the uncertain nature of the process by providing some way of capturing output ranges.

The above properties should be displayed by any model that claims to be a candidate representation of the required abstraction. They represent informal criteria against which such a model might be judged prior to formal evidence accumulation. Property P7 implies the need for some formality in any candidate model.

7

A Model

In this Chapter I present a candidate model, *KiTe*.

In the rest of this document, elements that are *KiTe* components are presented as *slanting text*.

7.1 Overview

Figure 7.1 shows a schematic overview of *KiTe*. The purpose is to introduce some terms and provide an introduction to some of the model components.

The first point to note is that I represent a *RealisedProcess*. The term ‘process’ is generally used to describe the technical aspects of software development only and is often used in a prescriptive way. A *prescriptive process* can be defined as “A description of a process that takes into account only technical aspects and implicitly makes assumptions that human factors do not affect process outcomes” (see Appendix A). For the intended model, we are interested in describing what actually happens during a software project and so must include the effects of project contexts. In order to avoid confusion between this meaning and the conventional use of the term ‘process’, I use the term *RealisedProcess*. I define a *RealisedProcess* as “A description of a process as it really happens i.e. that takes into account how all factors relevant to process outcomes, for example, the people involved and project contexts, affect these outcomes” (see

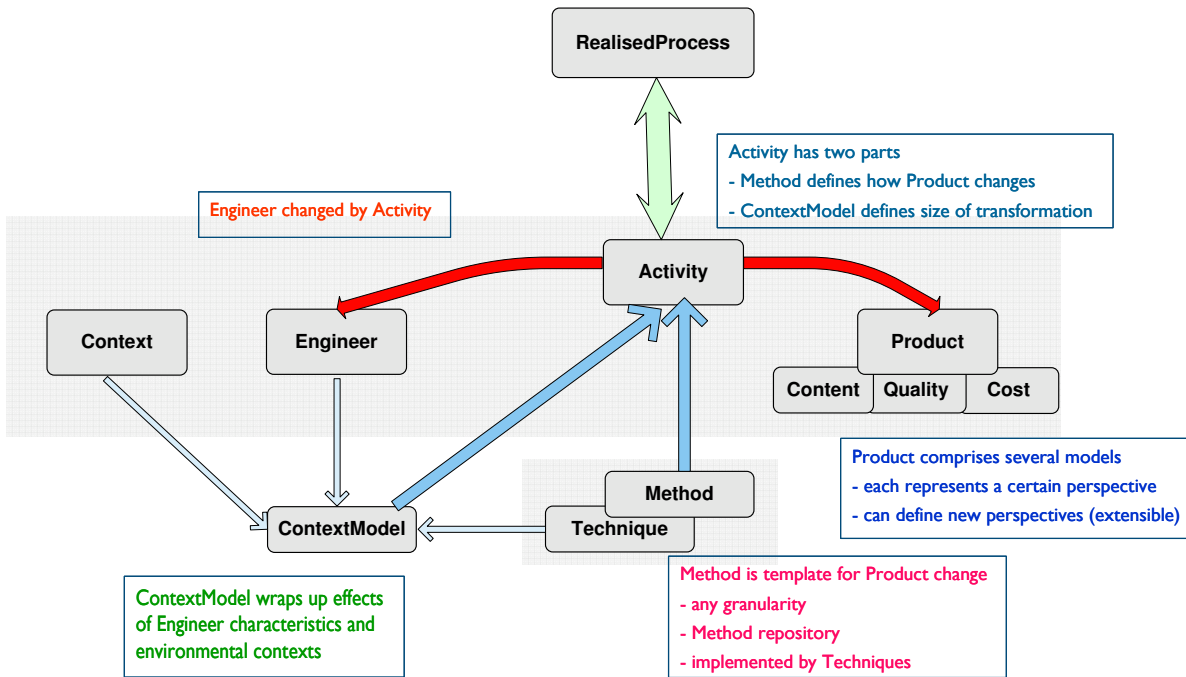


Figure 7.1: Schematic overview of KiTe

Appendix A).

The second point of note is that several of the properties presented in Section 6.2 state a need to represent alternatives. For example, property P3 indicates a need to represent a product using different measures and property P9 a need to represent different beliefs about how human factors affect outcomes. I address these criteria by proposing an abstraction that is a *framework* where each model component is a model in its own right. This allows modellers to choose the form of the framework model components that best represents their particular needs and beliefs. For example, if a project is to deliver software under specific quality and cost constraints, those constraints will be abstracted in a suitable way in the *Product Quality* and *Cost* models. If the process modeller believes that developer experience and skills are the only contextual factors that affect outcomes, these will be represented in the models for *Engineer* and *ContextModel*. As the industry matures and evidence becomes available to support specific models, these models become fixed within the *KiTe* framework as they are now current theories from which further hypotheses may be formed and tested in the context of the larger system. The framework thus provides support for formal experimentation.

The key aspects are overviewed below.

Product

Product is modelled as a set of states. I view these states as ‘pointing to’ the set of all possible states for all possible products. This allows a separation of the ‘product’ that is the subject of the *RealisedProcess* and a conceptual product in the real world and supports the situation described in Section 6.1.3. Figure 7.2 illustrates the concept. The first picture, a), represents all possible states of all possible products. The second, b), shows a subset of these states, those for the product with name ‘MyWebApp’. Only some of these apply in a particular project. For example, if ‘p’ is an upgrade project, all states of *Product* will include a representation of code content. The states for ‘p’ are shown in c). In d) I illustrate that only some attributes are relevant for project ‘p’. For example, the attributes depicted in yellow might represent ‘size’, ‘number of defects’, ‘cost in person hours’ and ‘maintainability’, but for project ‘p’ only ‘number of defects’ and ‘cost in person-hours’ apply.

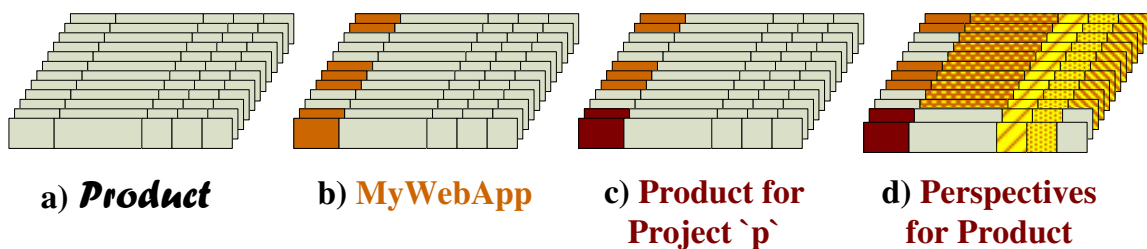


Figure 7.2: KiTe Product

Product represents all artifacts that describe the software being produced. These include the software delivered to the end customer and all requirements, designs, code, etc. to be delivered to the development organisation as an asset for use in later projects. *Product* attributes may be viewed via a number of *Perspectives*. Most commonly, these relate to the conventional product-related drivers i.e. *ContentPerspective* (‘how much is there’), *QualityPerspective* (‘how good is it’) and *CostPerspective* (‘how much did it cost’). However, I can encompass other product-related objectives, for example, the need to capture business value for product artifacts [20], by creating a new perspective model, for example *BusinessValuePerspective*. I also note that some factors generally described as ‘contextual’, for example, Boehm’s ‘required reusability’ (see Section 3.1) and Basili and Rombach’s ‘reliability requirements’ (see Section 4.2) are product-related objectives and are viewed in *KiTe* as attributes of *Product*.

Activity

A *KiTe Activity* encompasses both the task that is carried out (represented by *Method* and *Technique*) and how well the engineers carry out the task (represented by *ContextModel*). Both *Engineer* and *Product* are changed as a result of an *Activity*. For example, *Engineer* attributes, such as experience or skill, are likely to increase.

Method and Technique

A *KiTe Method* is defined as a set of transformations on *Product*. The transformation domain defines the *Method's Product*-related preconditions (for example, existence of some design content). The transformation is not constrained and this means that, in addition to the traditional tasks, for example, 'code from design documents', *KiTe* handles any task that causes change to *Product*. Tasks such as 'test first design', 'create a prototype based on a feature list' or 'code from prototype' are valid *Methods*. The definition of *Method* permits tasks of any granularity. So, for example, 'develop product from requirements' or even 'develop product' are as valid as 'carry out design review'.

A *Method* transformation has possibly many different codomain values for every domain value. For example, a *Method* that involves injecting defects into code may be defined as injecting 0 or more defects and so each domain value maps to one of a large number of possible values. *Method* may thus be considered as a family of transformations, or transformation template, that must be instantiated to provide a definitive transformation. *Technique* provides this instantiation and represents *how* a *Method* is carried out. For example, many believe that the technique of 'pair programming' yields better quality source than coding by a single person. In both cases, a change to source results, but the new actual values will probably be different in each case. *Technique* represents an 'average' of the results obtained when the *Technique* is applied in a large number of different circumstances.

I illustrate this concept in Figure 7.3. In the top two diagrams, I show that a *Method* applies to only some products. These are the products that have appropriate attributes (products D, E, I and K in the top diagram) and whose attributes have values that comply with the *Method's* precondition (the light green domain states in the middle diagram). In the bottom diagram, I show the many-many mappings provided by *Method* constrained to become functional ones by two *Techniques* (shown as red and blue mappings).

ContextModel

Researchers and practitioners have identified many factors that are believed to affect project outcomes. Some factors describe a 'match' between engineers and the product they are changing,

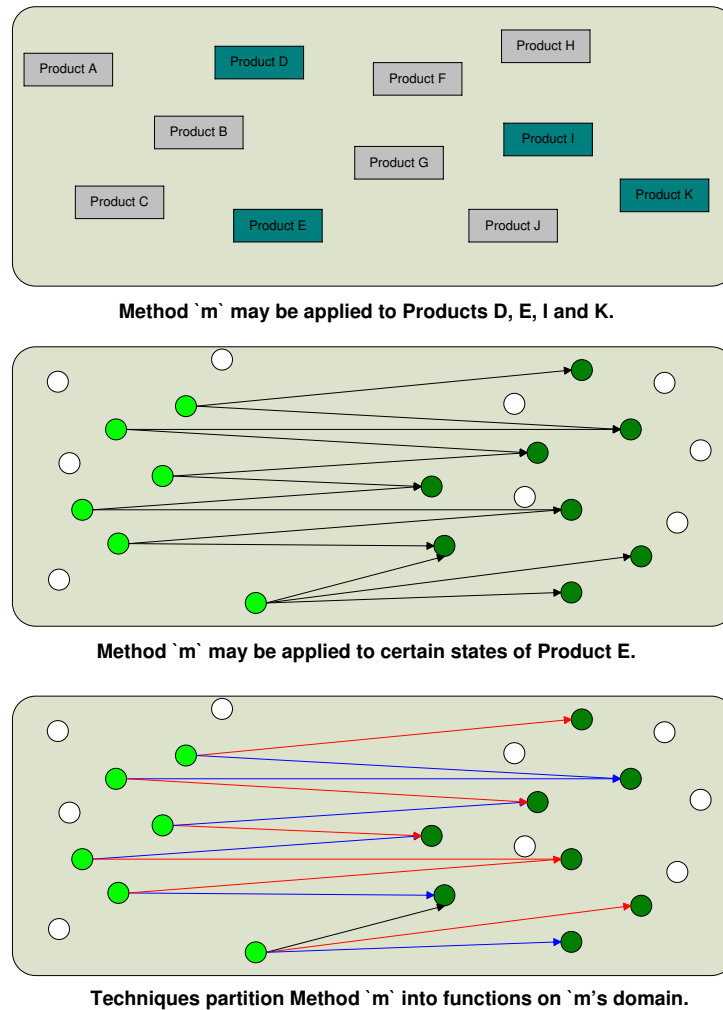


Figure 7.3: KiTe Method and Technique

for example, how familiar the engineer is with the subject area. Some relate to the *Techniques* they are using, for example, how much experience the engineer has with the *Technique* and whether or not appropriate tools are available. Many factors relate to the project environment. For example, project management may be supportive or overly demanding, expectations may be clearly defined or vague and engineers may be enthusiastic about the project or simply tired from overwork.

The list of possible factors is large and changes with time. This creates a number of problems. When creating models of the software development process, researchers must first choose which factors to include as input variables. This means the factors are now embedded in the model i.e. the model must be changed if new influencing factors are discovered. Researchers must also decide how to manage the large number. For example, some researchers implement a

rule-base, with resulting issues of scalability (see Section 4.3.1).

In a *KiTe RealisedProcess*, engineers change a product by carrying out *Methods* using *Techniques*. I suggest that handling of the factors described above can be simplified by viewing the factors as affecting how well the engineers are able to work i.e. by their effects on *engineer effectiveness*. In *KiTe*, I abstract all the factors that affect how engineers work into a *ContextModel*.

The key idea is that *ContextModel* represents engineer effectiveness by modifying the base effects of *Method* and *Technique* according to which factors it defines as relevant and how these are combined. For example, when capturing a waterfall process, the *ContextModel* is likely to be extremely simple, as contexts are generally ignored for this kind of process. A study in which engineer experience and skills are believed to be of relevance might involve only these factors and ignore all others. This is equivalent to stating that no other factors are believed to have influenced effectiveness. A more complex model might implement some ‘matching’ of skills with those required by *Product* or *Technique*. Such matching might be based on the Human Competencies model described in Section 4.5 i.e. by matching person and role capabilities.

Of course, for a model that aims to include large numbers of influencing factors, it is crucial that the abstraction for engineer effectiveness is such that complexity is reduced. I suggest that a suitable model will include a small number of orthogonal engineer-related characteristics. In Section 4.5, I described Curtis’s Layered Behavioral Model which suggests that individual capabilities are key for small projects and issues of uncertainty for larger projects. Based on these ideas, I propose that all contexts be mapped to the set ‘capability’, ‘certainty’ and ‘motivation’. In such an abstraction, for example, factors such as ‘large team’ and ‘threatened redundancy’ affect the values for ‘certainty’ and ‘motivation’.

ContextModel thus immediately allows existing beliefs about human-related contexts to be included when representing software development processes while providing an abstraction on which to base research into the effects of human factors on the process. In addition, context-related considerations are nicely partitioned from the rest of the model, and this enables researchers to easily ‘upgrade’ *ContextModel* when new knowledge is uncovered.

CapabilitySpec

In order to infer engineer effectiveness, it is likely that a *ContextModel* may require to carry out some matching of engineers to the product they are changing and the *Techniques* they are applying. For example, an engineer experienced in Pair Programming will probably be more effective applying a Pair Programming *Technique* than one who has no such experience.

Engineer skills and experiences and product and technique required skills are represented in *CapabilitySpecs*. *CapabilitySpec* thus serves to capture a set of capabilities. An engineer may

have experience in Java or skills with formal reviews. It may be that, to work with a product, knowledge of computer telephony or skills in C++ are required. A design technique may require expertise in structured design.

7.2 KiTe Model

In this section, I formally present *KiTe*. For each model component, I provide first a brief overview with some comment as to the rationale behind the component, follow this with a detailed description and then conclude with a description in formal notation. The reader may choose to omit the detailed and formal descriptions at first reading and ‘fill in the details’ at a later time. Before commencing, I discuss notations used.

Although *KiTe* is not a software product, for the detailed description I comply with the *IEEE Recommended Practice for Software Design Descriptions* [73], as this standard provides a suitable template for capture of systems comprising different kinds of elements. However, as the target system is an abstraction rather than a software implementation, some of the descriptions will be at a higher level than required for implementation. For example, when describing how *Activity* interfaces with *ContextModel*, implementation details, for example, ‘message passing’ or ‘function call’, are not applicable.

The Standard defines *design entity attributes Identification, Type, Purpose* (‘why it is there’), *Function* (‘what it does’), *Subordinates* (‘composed of’), *Dependencies* (‘uses’ or ‘requires the presence of’), *Interface* (‘how other entities interact with this’), *Resources* (‘external elements used’), *Processing* (‘rules for achieving function’) and *Data* (internal data elements). For *Type*, I categorise elements according to their main purpose in the system i.e. as of type *Aggregation* (main role is to contain other elements), *SetOfStates* (main role is to capture current state), *Projection* (main role is to project data from a *SetOfStates*), *DataStore* (main role is to hold attributes and values and *Transformation* (main role is to contribute to state change). I omit *Resources* as this is not relevant for modelling. For those elements that are models in their own right, the element will be described as a *Template* and some examples of processing and data are given. I use the ‘.’ notation for elements in a dependency relationship, for example, *Engineer.CapabilitySpec*.

For the formal description, I choose a representation that includes set constructs and state transitions. The *IEEE Recommended Practice for Software Requirements Specifications* [74] reminds us that “requirements methods and languages and the tools that support them fall into three general categories - object, process, and behavioral. Object-oriented approaches organize the requirements in terms of real-world objects, their attributes, and the services performed by those objects. Process-based approaches organize the requirements into hierarchies of func-

tions that communicate via data flows. Behavioral approaches describe external behavior of the system in terms of some abstract notion (such as predicate calculus), mathematical functions, or state machines.” Although the KiTe model is not a software system, the decision as to how best represent it may be approached by considering the above three alternatives. Woodcock and Davies, in their book “Using Z” [171] remind us that “Mathematical objects are often seen as collections of other objects: a square is a collection of points in a plane; a function is a collection of pairs linking arguments with values. These collections are called *sets*...”. As the problem space, i.e. the software development process, includes both structural aspects and transformational ones, I choose a representation that uses the basic mathematical notions of set constructs, with constructs that describe transformational elements captured as relations defined by state transitions.

KiTe components and the relationships between them are shown in Figure 7.4. I begin by recapping what is a *Project* and then overview some types I treat as ‘basic’, in that I use the types without providing any formal definition. I then address each *KiTe* component.

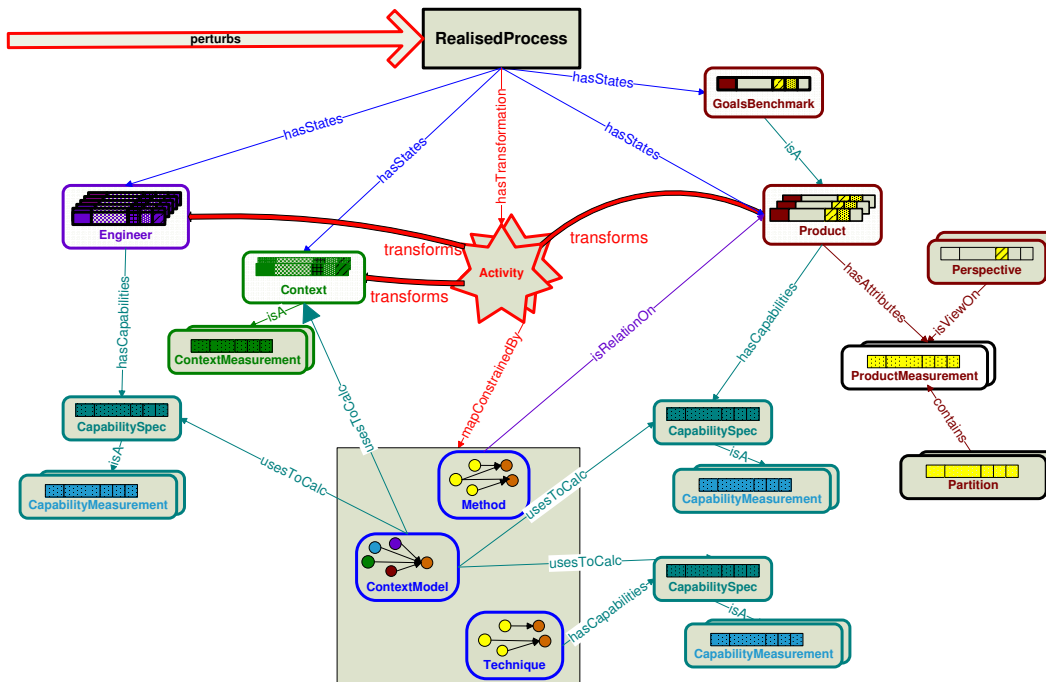


Figure 7.4: KiTe model relationships

7.2.1 Project

I define *Project* (Appendix A) as “... a temporary endeavour to create a unique service or product and with a definite beginning and end” [135]. As noted in Section 1.8, this definition says

nothing about the form of the service or product delivery and, in this thesis, I view a project as any effort that makes a delivery of any kind to any stakeholder. For example, a project might deliver a finished product to a customer, a prototype to the development group or a test plan to the test group.

As stated in Chapter 6, *KiTe* allows for the same conceptual product to be changed by many *Projects*. The description presented here allows for this while providing notation for elements in the context of a particular *Project*.

7.2.2 Atomic types

Overview

In this thesis, I treat some types as ‘atomic’ and do not define them in detail. For example, *KiTe* includes several identifiers, including *ProductIdentifier* and *EngineerIdentifier*. *ProductIdentifier* provides a way of uniquely identifying the product that is changed during a project and *EngineerIdentifier* uniquely identifies the engineers involved in the project. These types are introduced as required. *KiTe* also includes several types that describe *measurements*. A *measurement* is a representation of some attribute of interest that includes some notion of attribute identification, meaning, scale and value. Types include *ProductMeasurement*, which represents some product-related attribute, and *ContextMeasurement*, which represents some context-related attribute. As discussed in Section 3.4.1, the measurement of software-related attributes is problematic for several reasons, including the incorrect use of measurement scales in common software engineering practice. Further consideration of this is outside the scope of this thesis and later in this thesis I simply represent a *Measurement* as a binary relation that represents an attribute-value pair. For example, (Java, High) indicates some attribute called ‘Java’ with value ‘High’. A related concept is that of *measure*. A *measure* is a *measurement* with no value assigned — a kind of ‘measurement template’. *KiTe* includes *ProductMeasure*.

7.2.3 Product

Overview

Product is an abstraction of the deliverables from a *RealisedProcess*.

I model *Product* as a set of states i.e. *Product* represents the set of all states that describe the product for a *RealisedProcess*. *Product* includes a *ProductIdentifier* that identifies the conceptual product in the real world, for example, ‘MyWebApp v.1.6’. This identifier is unique to *Product* for a *RealisedProcess*. *Product* also has a *CapabilitySpec* that specifies the kinds of capabilities required for working with the conceptual product, for example, ‘experience with

Java'. *Product* also includes some attributes that describe the dynamic status of attributes-of-interest, for example, 'size' and 'number of defects'. These attributes are represented as a set of *ProductMeasurement*.

Detailed description

Type SetOfStates Template.

Purpose Abstract the product-related deliverables for a project for the purpose of tracking status.

Function *Product* describes the characteristics and dynamic status of the software being produced.

Subordinates None.

Dependencies *ProductIdentifier* ('isIdentifiedBy'); *CapabilitySpec* ('hasCapabilities'); *ProductMeasurement* ('hasAttributes').

Interface *GoalsBenchmark* ('isA'); *RealisedProcess* ('hasStates'); *Activity* ('transforms'); *Method* ('isRelationOn').

Processing None.

Data *Product* references a subset of the 'global' product state space i.e. the space containing all possible states of all possible conceptual products. A *Product* state comprises *ProductIdentifier* which identifies the product in the outside world (e.g. 'MyWebApp v.1.6'), *CapabilitySpec* containing information about the capabilities required for working with this *Product* and a set of *ProductMeasurement* capturing the dynamic status of *Product*. *ProductIdentifier*, *CapabilitySpec* and *ProductMeasurement* are references to specific states in the 'global' state space.

Formal description

Product is represented as a set of states. I represent the set of all possible states of all possible products as \mathcal{PS} and the set of states for a specific *RealisedProcess* rp as PS^{rp} .

$$PS^{rp} \subseteq \mathcal{PS} \quad (7.1)$$

A *Product* state can belong to a single *RealisedProcess* only i.e. the intersection of *Product* states for different *RealisedProcesses* is empty.

$$\forall rp1, rp2 \in \mathcal{RP} \quad rp1 \neq rp2 \Leftrightarrow (PS^{rp1} \cap PS^{rp2} = \emptyset) \quad (7.2)$$

A *Product* state ps includes a *ProductIdentifier* that identifies the real-world product to which ps applies. Function pid projects *ProductIdentifier* from $ps \in \mathcal{PS}$.

\mathcal{PID} is the set of all *ProductIdentifier*.

$$pid : \mathcal{PS} \mapsto \mathcal{PID} \quad (7.3)$$

RealisedProcess has the properties that a *ProductIdentifier* is unique to a single $rp \in \mathcal{RP}$ and that all product states for a *RealisedProcess* rp have the same value for *ProductIdentifier*. The first property provides a stronger statement of Equation 7.2 i.e. it is the value of the state's *ProductIdentifier* attribute that associates the state with a single *RealisedProcess*.

$$\forall rp1, rp2 \in \mathcal{RP}, \forall ps1 \in PS^{rp1} \forall ps2 \in PS^{rp2} \quad (pid(ps1) = pid(ps2) \Leftrightarrow PS^{rp1} = PS^{rp2}) \quad (7.4)$$

A *Product* state includes a *CapabilitySpec* that captures capabilities required for working with the product. Function $pcps$ projects *CapabilitySpec* from *Product*.

$$pcps : \mathcal{PS} \mapsto \mathcal{CPS} \quad (7.5)$$

CapabilitySpec for PS^{rp} may change as rp progresses, for example, if the requirement to deliver on a specific operating system changes, indicating the need to add a new required capability. This means that, although each product state maps to a single *CapabilitySpec*, the projected *CapabilitySpec* for *Product* for *RealisedProcess* rp may contain more than one element i.e. is the set of *CapabilitySpec* for all states $ps \in PS^{rp}$.

$$pcps(PS^{rp}) = \{pcps(ps) \mid ps \in PS^{rp}\} \quad (7.6)$$

A *Product* state includes the status of the product-related attributes of interest for rp , for example, 'number of defects'. Each attribute-of-interest is represented as a *ProductMeasurement*.

$\mathcal{PM}\mathcal{E}$ is the set of all possible *ProductMeasurement*. Relation pme projects the set of *ProductMeasurement* for *Product* state $ps \in \mathcal{PS}$. This set represents the values of the product-related attributes-of-interest for the state and belongs to the power set of *ProductMeasurement*.

$$pme : \mathcal{PS} \mapsto 2^{\mathcal{PM}\mathcal{E}} \quad (7.7)$$

Each *ProductMeasurement* is based on a *ProductMeasure*. The *ProductMeasure* represents the attribute-of-interest without the value of the attribute i.e. it is a kind of *ProductMeasurement* template.

\mathcal{PM} is the set of all *ProductMeasure*. Function pm extracts the *ProductMeasure* from a

ProductMeasurement. This enables the isolation of a specific attribute from a set and subsequent access to the attribute's value.

$$pm : \mathcal{PME} \mapsto \mathcal{PM} \quad (7.8)$$

7.2.4 CapabilitySpec

Overview

CapabilitySpec represents a set of engineer capabilities, for example, skills, or the set of capabilities required for working with a specific product or technique. I note that capabilities are not limited to the commonly used 'skills' and 'experience' but may represent, for example, the 'behavioural competencies' such as 'Sociability' described by Acuna and Juristo (see Section 4.5).

I model *CapabilitySpec* as a set of *CapabilityMeasurement* i.e. as a subset of the set of all possible combinations of all possible values of all possible *CapabilityMeasurement*.

Detailed description

Type DataStore.

Purpose Required for matching of engineer capabilities to product and technique required capabilities.

Function Captures descriptive characteristics, technologies and experiences relating to the parent element. *Product.CapabilitySpec* characterises the skills required for working with *Product*, for example, subject area description, required implementation technologies, etc. *Engineer.CapabilitySpec* captures *Engineer* capabilities, for example, experience with Java or level of extroversion. *Technique.CapabilitySpec* characterises the skills required for working with *Technique*.

Subordinates None.

Dependencies *CapabilityMeasurement* via 'isA' relationship.

Interface *Product*, *Engineer* and *Technique* via a 'hasCapabilities' relationship; *ContextModel* interfaces with *Product.CapabilitySpec*, *Engineer.CapabilitySpec* and *Technique.CapabilitySpec* via a 'usesToCalc' relationship.

Processing None.

Data A *CapabilitySpec* references a set of *ProductMeasurement*.

Formal description

The set of all possible *CapabilitySpec* is \mathcal{CPS} . The set of all possible *CapabilityMeasurement* is $\mathcal{CM}\mathcal{E}$.

A *CapabilitySpec* is a set of *CapabilityMeasurement* i.e. is a set that is one of the power set of the set of all possible *CapabilityMeasurement*. For example, a *CapabilitySpec* might be the set $\{(\text{Java}, \text{High}), (\text{Design}, 6), (\text{Linux}, \text{Medium})\}$, indicating capabilities in ‘Java’, ‘Design’ and ‘Linux’ with appropriate values.

$$\mathcal{CPS} = 2^{\mathcal{CM}\mathcal{E}} \quad (7.9)$$

7.2.5 Partition

Overview

Partition serves to group product-related attributes into non-overlapping sets that represent different kinds of product descriptions. *Partitions* must be non-overlapping because they play a key role in the definition of *Method* and *Technique* preconditions and effects. Each *Partition* contains attributes that describe one kind of product description and *Methods* and *Techniques* are defined by the kinds of descriptions they change. *Partitions* are *Definition*, *Architecture*, *Design*, *Source*, *Integration* and *Packaged*. The decision is based on the need to consider all stakeholders when defining what comprises *Product* i.e. the need to abstract the content of all artifacts that capture any aspect of the software that will be delivered to the end customer. The chosen *Partitions* represent the descriptions most frequently found in the literature and would seem to cover all possibilities. For example, *Definition* includes anything that addresses the problem to be solved and so might include formal requirements, feasibility studies and XP stories. *Source* includes, in addition to code, data files and document sources. *Packaged* refers to the system as ready to deliver. For small projects, this might be the same as *Integration* i.e. the ‘packaging’ step is ‘free’ as no additional work is required after the system is integrated. For larger projects, this is almost certainly not the case, for example, if many large files must be copied from various locations to a ‘delivery’ location.

Note that attributes must be specified for each *Partition*. For example, ‘number of known defects in requirements’ and ‘number of known defects in code’ are two separate attributes and are represented in different *Partitions*.

Detailed description

Type Projection.

Purpose Provide a means of categorising product attributes such that attributes of interest to all stakeholders are included. For example, information about architectures may be of interest to the developing organisation.

Function Project the product attributes and values for a specific kind of product information, for example. those describing architectures.

Subordinates None.

Dependencies *PartitionIdentifier* ('isIdentifiedBy'); *ProductMeasurement* ('contains').

Interface None.

Processing None.

Data *Partition* has a *PartitionIdentifier* that is one of *DefinitionPartition*, *ArchitecturePartition*, *DesignPartition*, *SourcePartition*, *IntegrationPartition* and *PackagedPartition* and a set of *ProductMeasurement*.

Formal description

The set of *ProductMeasurement* is partitioned into six non-overlapping subsets, called *Partitions*. These describe different representations of the product. For example, representations that relate to problem definitions such as requirements-related attributes are included in the 'Definition' partition. *Partitions* are identified as *DEFNP*, (*DefinitionPartition*), *ARCHP*, (*ArchitecturePartition*), *DESNP*, (*DesignPartition*), *SRCEP*, (*SourcePartition*), *INTGP*, (*IntegrationPartition*) and *PACKP*, (*PackagedPartition*). This set of *PartitionIdentifier* is *PAID*.

$$\mathcal{PAID} = \{\mathit{DEFNP}, \mathit{ARCHP}, \mathit{DESNP}, \mathit{SRCEP}, \mathit{INTGP}, \mathit{PACKP}\} \quad (7.10)$$

Relation *ppame* projects the set of *ProductMeasurement* for *Product* state $ps \in \mathcal{PS}$ and *PartitionIdentifier* $paid \in \mathcal{PAID}$.

$$ppame: (\mathcal{PS} \times \mathcal{PAID}) \mapsto 2^{\mathcal{PM}\mathcal{E}} \quad (7.11)$$

For all *Product* states, a *ProductMeasurement* belongs to only one *Partition* and all *ProductMeasurement* belong to some *Partition*.

$$\forall ps \in \mathcal{PS}, \forall paid1, paid2 \in \mathcal{PAID} \quad paid1 \neq paid2 \Leftrightarrow (ppame(ps, paid1) \cap ppame(ps, paid2)) = \emptyset \quad (7.12)$$

$$\forall ps \in \mathcal{PS}, \forall me \in pme(ps), \exists paid \in \mathcal{PAID} \quad (me \in ppame(ps, paid)) \quad (7.13)$$

7.2.6 Perspective

Overview

For *Product*, the product-related attributes of interest are viewed in a way that is meaningful to the modeller, for example, those describing some aspect of quality might be viewed together. Such a view is a *Perspective*. A *Perspective* effectively constrains which attributes are to be viewed. Different stakeholders may choose to view *Product* in different ways, i.e. the choice of *Perspectives* is flexible and unrestricted.

Perspective is represented by a set of *ProductMeasures*. The *Perspective* overlays on *Product* to project on the required set of attributes.

I model *Perspective* as a single state in the state space of all possible combinations of all possible values of all possible *ProductMeasure*.

Detailed description

Type Projection.

Purpose Represent a particular view of product attributes and values as these change throughout the project.

Function Projects attributes and values for a product-related objective, for example, *Content* or *Quality*.

Subordinates *ProductMeasure*.

Dependencies *ProductMeasurement* ('isViewOn').

Interface None.

Processing The set of *ProductMeasure* provides a view on the attributes-of-interest for *Product*, as represented by *Product's* set of *ProductMeasurement*.

Data *Perspective* is represented as a set of *ProductMeasure*.

Formal description

\mathcal{PE} is the set of all possible *Perspective*. The set of *Perspective* for *RealisedProcess* $rp \in \mathcal{RP}$ is PE^{rp} .

A *Perspective* is a set of *ProductMeasure* i.e. is a set that is one of the power set of the set of all possible *ProductMeasure*.

$$\mathcal{PE} = 2^{\mathcal{PM}} \quad (7.14)$$

Relation $ppeme$ projects the set of *ProductMeasurement* for a *Product* state $ps \in \mathcal{PS}$ and *Perspective* $pe \in \mathcal{PE}$.

$$ppeme: (\mathcal{PS} \times \mathcal{PE}) \mapsto 2^{\mathcal{PM}} \quad (7.15)$$

For all *Product* states in *RealisedProcess* rp , a *ProductMeasurement* belongs to only one *Perspective* and all *ProductMeasurement* belong to some *Perspective*.

$$\forall rp \in \mathcal{RP}, \forall ps \in PS^{rp}, \forall pe1, pe2 \in PE^{rp} \quad pe1 \neq pe2 \Leftrightarrow (ppeme(ps, pe1) \cap ppeme(ps, pe2) = \emptyset) \quad (7.16)$$

$$\forall rp \in \mathcal{RP}, \forall ps \in PS^{rp}, \forall me \in pme(ps), \exists pe \in PE^{rp} \quad (me \in ppeme(ps, pe)) \quad (7.17)$$

The set of *ProductMeasurement* for a *Product* state $ps \in PS^{rp}$ can be found via *Partitions* or *Perspectives*.

$$\forall rp \in \mathcal{RP}, \forall ps \in PS^{rp} \quad pme(ps) = \{ppeme(ps, pe) \mid pe \in PE^{rp}\} \quad (7.18)$$

$$\forall rp \in \mathcal{RP}, \forall ps \in PS^{rp} \quad pme(ps) = \{ppame(ps, paid) \mid paid \in \mathcal{PAID}\} \quad (7.19)$$

7.2.7 GoalsBenchmark

Overview

For many projects, there is an agreement with the customer that delivery of the product will occur when a certain quality level is reached, for example, when all known defects after testing are minor and will not affect product functionality. For internal projects, for example, to deliver a test plan to project management, there will probably be an agreement with the internal customer about the criteria for delivery, for example, ‘when complete’ or ‘in 2 weeks’. A *GoalsBenchmark* represents the expected *Product* state at *RealisedProcess* completion. This is represented as a subset of \mathcal{PS} . It is probable that this subset has more than one possible state because the expectation will be of the form, for example, ‘number of known defects less than 10’ i.e. the set of ten states with ‘number of known defects’ equal to 9, 8, 70.

Detailed description

Type SetOfStates.

Purpose Capture the agreed status for the product-related objectives for a project at time of delivery.

Function As for *Product*.

Subordinates As for *Product*.

Dependencies *Product* ('isA').

Interface Interfaced to by *RealisedProcess* ('hasStates').

Processing None.

Data As for *Product*.

Formal description

$GB^{rp} \subseteq \mathcal{PS}$.

7.2.8 Engineer**Overview**

Engineer is an abstraction of the people involved in causing change to *Product* in a *Realised-Process*.

I model *Engineer* as a set of states i.e. *Engineer* represents the set of all states that describe the engineers for a *RealisedProcess*. Engineers represented include those in the roles of analysts (who change definitions and architectures), designers and coders (who change designs and sources), technical writers (who produce documents that are included in the packaged product) and test personnel (who provide information about product quality). *Engineer* does not include, for example, project managers as they do not directly change *Product*.

Engineer includes a set of *EngineerIdentifier* that identifies the real-world engineers, for example, 'John Smith' and 'Jane Doe'. *Engineer* also has a set of *CapabilitySpec* that specifies the engineers' capabilities, for example, 'experience with Java' or 'extroverted'.

Detailed description

Type SetOfStates.

Purpose Abstract the people involved in change to *Product*, for example, analysts, architects, coders and build personnel.

Function *Engineer* describes the dynamic status of the people involved in changing *Product*.

Subordinates None.

Dependencies *EngineerIdentifier* ('identifiedBy'); *CapabilitySpec* ('hasCapabilities').

Interface *RealisedProcess* ('hasStates'); *Activity* ('transforms').

Processing None.

Data *Engineer* references the 'global' engineer state space i.e. the space containing all possible states of all possible engineers. An *Engineer* state comprises a set of *EngineerIdentifier* that identifies the real-world engineers and a set of *CapabilitySpec* containing information about the capabilities of the engineers. The specific model defines which capabilities are included.

Formal description

Engineer is represented as a set of states. I represent the set of all possible states of all possible engineers as \mathcal{ES} and the set of states for *RealisedProcess* rp as ES^{rp} .

$$ES^{rp} \subseteq \mathcal{ES} \quad (7.20)$$

An *Engineer* state es includes a set of *EngineerIdentifier* that identifies the real-world engineers to which es applies. Function eid projects the set of *EngineerIdentifier* from $es \in \mathcal{ES}$.

\mathcal{EID} is the set of all *EngineerIdentifier*.

$$eid : \mathcal{ES} \mapsto 2^{\mathcal{EID}} \quad (7.21)$$

The set of *EngineerIdentifier* for *RealisedProcess* rp is

$$eid(ES^{rp}) = \{eid(es) \mid es \in ES^{rp}\} \quad (7.22)$$

An *Engineer* state includes a set of *CapabilitySpec* that capture capabilities possessed by the engineers. Relation $ecps$ projects the *CapabilitySpec* for *EngineerIdentifier* for an *Engineer* state.

$$ecps : \mathcal{ES} \times \mathcal{EID} \mapsto \mathcal{CPS} \quad (7.23)$$

CapabilitySpec for a real-world *engineer* may change as rp progresses, for example, as the engineer gains experience. This means that, although each engineer state maps to a single *CapabilitySpec* for each engineer, the engineer's projected *CapabilitySpec* for *RealisedProcess* rp may contain more than one element i.e. is the set of *CapabilitySpec* for all states $es \in ES^{rp}$. The set of *CapabilitySpec* for *EngineerIdentifier* id in *RealisedProcess* rp is

$$ecps(ES^{rp}, id) = \{ecps(es, id) \mid es \in ES^{rp} \wedge id \in eid(es)\} \quad (7.24)$$

7.2.9 Context

Overview

Context is an abstraction of the non-engineer-related factors that are believed to affect how well engineers change *Product* in a *RealisedProcess*. Such factors may include those both internal to the project, for example, project management style, and factors resulting from interfaces with other projects (for example, communication issues) or groups (for example, impending company merger).

I model *Context* as a set of states i.e. *Context* represents the set of all states that describe the project-specific factors for a *RealisedProcess*. Characteristics of the engineers themselves are excluded. Examples might include ‘company about to be taken over’, ‘poor development environment’ and ‘culture of overworking employees’. A *Context* is realised as a set of *ContextMeasurement*.

Detailed description

Type SetOfStates.

Purpose Abstract the project environment.

Function Captures non-engineer-related aspects of the project environment believed to have an effect on product outcomes. Examples are ‘availability of customer’, ‘tool support’.

Subordinates None.

Dependencies *ContextMeasurement* (‘isA’).

Interface *RealisedProcess* (‘hasStates’); *ContextModel* (‘usesToCalc’).

Processing None.

Data *Context* references the ‘global’ context state space i.e. the space containing all possible states of all possible contexts. *Context* attributes and values are represented by a set of *ContextMeasurement*. The specific model defines which items are included.

Formal description

Context is represented as a set of states. I represent the set of all possible states of all possible contexts as CS and the set of states for *RealisedProcess* rp as CS^{rp} .

$$CS^{rp} \subseteq CS \quad (7.25)$$

A *Context* state includes a set of *ContextMeasurement* that represents the context. \mathcal{CME} is the set of all possible *ContextMeasurement*. Relation cme projects the set of *ContextMeasurement* for *Context* state $cs \in \mathcal{CS}$.

$$cme: \mathcal{CS} \mapsto 2^{\mathcal{CME}} \quad (7.26)$$

7.2.10 Method

Overview

Method represents a set of transformations on *Product*. It provides a definition of which attributes of *Product* change during the transformation and a description of how they change. It thus also provides a precondition on *Product* attributes, for example, ‘all requirements must be represented in designs’. *Method* effectively provides a ‘template’ for change that is unambiguous. For example, a ‘code from designs’ *Method* has a precondition relating to existence of designs and causes increase in attributes describing source content and source quality. If the *Method* is ‘code from designs and fix design and code defects’, the precondition will be the same, but the attributes that change will include design-related attributes. The purpose of *Method* is thus to provide clarity about what is changing in *Product* when some task is carried out. There will be many possible ways to make these changes.

Detailed description

Type Transformation.

Purpose Capture a development task in an unambiguous way. Include tasks that do not change *Product*.

Function Captures a task as a set of possible transformations on *Product*. The domain for the transformations defines acceptable precondition states of *Product*. This means that *Product* must contain the correct attributes in its *ProductMeasurements* and that these attributes must have appropriate values. The codomain represents all possible outcomes of the task.

Subordinates None.

Dependencies *Product* (‘isRelationOn’); *Technique* (‘mapConstrainedBy’).

Interface None.

Processing None.

Data Binary, many-many relation on *Product*.

Formal description

Method is a transformation of states of *Product* i.e. is a binary relation $\mathcal{PS} \times \mathcal{PS}$. The set of all possible *Method* is \mathcal{M} .

$$\mathcal{M} = \mathcal{PS} \times \mathcal{PS} \quad (7.27)$$

For any *Method* $m \in \mathcal{M}$, the relation defined by m has many possible domain states. For example, for a *Method* involving designing a product from some architectures, the domain states for m include all *Product* states that represent a product containing the relevant architectures. This might include states that represent, for example, a product with or without requirements or existing integrations.

The set of possible domain states $\mathcal{D}(m)$ for m defines m 's precondition. For *RealisedProcess* rp , the domain for m is restricted to those states in PS^{rp} , i.e. the states with PS^{rp} 's *ProductIdentifier*. This set of domain states is defined by mapping m_{pre} .

$$\forall m \in \mathcal{M}, \forall rp \in \mathcal{RP} \quad m_{pre}(m, rp) = \mathcal{D}(m) \cap PS^{rp} \quad (7.28)$$

A domain element for a *Method* maps to many possible values. These values represent the possible transformations for *Method*. For example, a *Method* that represents creating designs from architectures may be characterised by an increase in the value of a 'design' attribute along with an increase in the value of a 'design defects' attribute. For a *Product* domain state with both 'designs' and 'design defects' values equal to zero, the possible mappings include many states i.e. those with 'designs' > 0 and 'design defects' > 0 . A *Method* may thus be viewed as a 'mapping template'.

I now consider the standard relational properties of *reflexion*, *symmetry* and *transitivity* [159] with respect to *Method*. The reason is one of exploration — I would like to investigate other possible aspects of *Method* not immediately apparent, for example equivalences and partial orderings.

Properties of reflexion include *reflexive* and *irreflexive*. A relation is reflexive if every domain element maps to itself. It is irreflexive if a single domain element does not map to itself. For *Method*, I allow reflexion i.e. one of the possible transformations of m represents no change in *Product* state. This is required for *Activities* that do not change *Product*, for example, developer design discussions.

Properties of symmetry include *symmetric*, *antisymmetric* and *asymmetric*. A relation, R , is symmetric if and only if aRb implies bRa . R is antisymmetric if the inclusion of both aRb and bRa is possible only if $a = b$. An asymmetric relation does not permit the antisymmetric case i.e. both aRb and bRa may never be true. *Method* changes *Product* in a way that is consistent

with antisymmetry i.e. a transformation and its inverse occur only in the case of no change to *Product* state.

Properties of transitivity include *transitive* and *atransitive*. A relation R is transitive if the inclusion of both aRb and bRc imply the inclusion of aRc . Because *Method* defines ‘change templates’ rather than defining size of change, the transitivity property holds. For example, the ‘design’ *Method* introduced above results in an increase or decrease in value of attributes that represent designs and an increase in value of attributes that represent numbers of design defects. If the defects element of the mapping includes (0,6) and (6,30), the implication is that (0,30) will also be a possibility. The transitivity property holds.

Method is reflexive, antisymmetric and transitive. *Method* thus represents a *weak partial order* relation on \mathcal{PS} [159].

7.2.11 Technique

Overview

In the previous Section, I described *Method* as a set of transformations on *Product* that effectively defines what is changed in *Product*. I noted that *Method* is effectively a template and that there would be many different ways of carrying out the *Method*. *Technique* describes *how* *Product* change is made and represents one way of implementing *Method*. For example, for *Method* ‘DesignAndCode’, possible *Techniques* might be ‘PairProgramming’ and ‘ImplementOOArchitecture’. Each would result in the same kind of change to the same *Product Partitions*, for example, increase in *Design* and *Source* attributes. However, each would have a slightly different outcome, for example, many believe that ‘PairProgramming’ will result in a smaller increase in defect numbers. *Technique* thus constrains the *Method* relation to a functional one. The resulting *Product* transformation may be regarded as the ‘expected’ transformation for this *Technique* i.e. the average result for the *Technique* when applied to many different kinds of products by many different kinds of engineers in many different kinds of environments.

Technique also has a *CapabilitySpec* that describes the capabilities required for working with the *Technique*. For example, an ‘OO design’ *Technique* requires skills in OO design.

Detailed description

Type Transformation.

Purpose Capture changes to *Product* when a specific *Method* is carried out in a particular way.

Function Constrains the many-many transformation on *Product* defined by a *Method* to a functional (many-one) transformation. The domain for *Technique* includes that of its

associated *Method*. Each domain state maps to a specific value that represents an ‘average’ result. The value may be absolute (for example, an ‘expected’ defects injection density) or relative (for example, when *KiTe* is used to compare outcomes against some benchmark).

Subordinates None.

Dependencies *CapabilitySpec* (‘hasCapabilities’).

Interface *Method* via ‘mapConstrainedBy’ relationship.

Processing Extracts subset of *Product* that matches ‘Precondition’ and outputs the states of *Product* to which each maps.

Data An *Identifier* identifies the *Technique*. Transformations are represented as a binary (functional) relation on *Product*.

Formal description

Technique is a functional mapping between states of *Product* along with a description of the capabilities required to work with the *Technique*. The set of all possible *Technique* is \mathcal{T} .

Function *tcps* projects the *CapabilitySpec* for a *Technique*.

$$tcps: \mathcal{T} \mapsto \mathcal{CPS} \quad (7.29)$$

Relation *ttran* projects the functional mapping on *Product* for a *Technique*.

$$\forall t \in \mathcal{T} \quad ttran(t): (\mathcal{PS} \mapsto \mathcal{PS}) \quad (7.30)$$

As for *Method*, the transformation *ttran* defined for *Technique* *t* has a number of possible domain states and these represent *t*’s precondition. The set of domain states for *RealisedProcess* *rp* is defined by mapping *tpre*.

$$\forall t \in \mathcal{T}, \forall rp \in \mathcal{RP} \quad tpre(t, rp) = \mathcal{D}(ttran(t)) \cap \mathcal{PS}^{rp} \quad (7.31)$$

A *Technique* may be applied to a *Method* *m* if all domain states for *m* are included in *t*’s domain. This means that every precondition state defined by *m* is included in the functional mapping defined by *t*. The set of *Technique* that can be applied to a *Method* is $T^{m \in \mathcal{M}}$.

$$T^{m \in \mathcal{M}} = \{t \mid t \in \mathcal{T} \wedge \mathcal{D}(m) \subseteq \mathcal{D}(ttran(t))\} \quad (7.32)$$

A *Technique* $t \in T^{m \in \mathcal{M}}$ may contain domain states not included in *m*’s domain. For example, *t* might include mappings for states with architectural content whereas *m* might be

defined for zero architectural content only. The result of t applied to m is thus the transformation defined by t restricted to the m 's domain elements. I denote 't applied to m' as $t \circ m$.

$$\forall t \in \mathcal{T} \forall m \in \mathcal{M} \quad t \circ m = ttran(t) \cap m \quad (7.33)$$

The relation representing a *Technique*'s transformation may apply to many *Methods* and this means that, in the real world, the *Technique* may be implemented to realise any of these *Methods*.

I now check for reflexion, symmetry and transitivity. As for *Method*, I allow reflexion i.e. one of the possible transformations of $t \in \mathcal{T}$ represents no change in *Product* state. The antisymmetry property holds i.e. for T , aTb and bTa both are included only if $a = b$. *Technique* is atransitive i.e. as a function, inclusion of all of (x,y), (y,z) and (x,z) is not possible. *Technique* is reflexive, antisymmetric and atransitive.

Further consideration of function properties leads to the understanding that *Technique* is *one-one* (domain elements will change in a consistent way) but it is not *onto* (all codomain elements i.e. PS are not included in the mapping). This means that we may not discuss inverse transformations i.e. we can not 'work backwards' through a set of *Techniques* applied in composition.

7.2.12 ContextModel

Overview

The role of *ContextModel* is to abstract the effects of the project environment on *RealisedProcess* outcomes. *ContextModel* has two responsibilities. The first is to represent how well specific engineers are able to carry out a *Technique* on a given *Product*. *ContextModel* 'matches' engineer capabilities with those required for working with the *Technique* and *Product*. This representation causes the transformation on *Product* defined by *Technique* alone to be altered to effect a different transformation. The second responsibility is to effect change to *Engineer* and *Context* as a result of engineer involvement with *Technique* and *Product*. For example, carrying out designs for a computer telephony product may increase engineer knowledge about computer telephony and some believe that engineers carrying out 'PairProgramming' become more satisfied.

ContextModel uses information about the various capabilities and contexts it requires for matching and thus has an expectation about the form of these. There is thus close-coupling between *ContextModel* and models for *CapabilitySpec* and *Context* i.e. these are all part of a single 'human factors' model. This model represents researchers' beliefs and provides a way of making visible assumptions in studies involving models.

Detailed description

Type Transformation Template.

Purpose Abstract the effects of the project-specific environment on the *RealisedProcess*.

Function Matches characteristics of the *Activity* (i.e. *Context*, *Engineers*, *Product* and *Technique*) to provide a *Product* transformation adjustment and to transform *Engineer* and *Context*.

Subordinates None.

Dependencies *Context* ('usesToCalc'); *Engineer.CapabilitySpec* ('usesToCalc'); *Product.CapabilitySpec* ('usesToCalc'); *Technique.CapabilitySpec* ('usesToCalc').

Interface *Activity* ('mapConstrainedBy').

Processing None.

Data Ternary relation on *Product* (original transformation and new end states). Binary relation on *Engineer*. Binary relation on *Context*.

Formal description

ContextModel may be described as a relation between various *RealisedProcess* elements. The relation defines the elements that are inputs to *ContextModel* and those that are outputs. Domain elements are $(\mathcal{PS} \times \mathcal{PS})$, \mathcal{ES} , \mathcal{CS} and \mathcal{T} . Codomain elements are \mathcal{PS} , \mathcal{ES} and \mathcal{CS} .

The set of all *ContextModel* is \mathcal{CM} .

$$\mathcal{CM} = (\mathcal{PS} \times \mathcal{PS} \times \mathcal{ES} \times \mathcal{CS} \times \mathcal{T}) \times (\mathcal{PS} \times \mathcal{ES} \times \mathcal{CS}) \quad (7.34)$$

Relation $cmprod$ provides a new end state for a transformation on $ps \in \mathcal{PS}$ according to *Context*, *Product* and *Technique* specifics and *Engineer* capabilities.

$$cmprod: (\mathcal{PS} \times \mathcal{PS}) \times \mathcal{ES} \times \mathcal{CS} \times \mathcal{T} \mapsto \mathcal{PS} \quad (7.35)$$

Relation $cmeng$ effects a transformation on $es \in \mathcal{ES}$ according to *Context*, *Product* and *Technique* specifics and *Engineer* capabilities.

$$cmeng: \mathcal{PS} \times \mathcal{ES} \times \mathcal{CS} \times \mathcal{T} \mapsto \mathcal{ES} \quad (7.36)$$

Relation $cmcontext$ effects a transformation on $cs \in \mathcal{CS}$ according to *Context*, *Product* and *Technique* specifics and *Engineer* capabilities.

$$cmcontext: \mathcal{PS} \times \mathcal{ES} \times \mathcal{CS} \times \mathcal{T} \mapsto \mathcal{CS} \quad (7.37)$$

7.2.13 Activity

Overview

Activity effects change to the state of a *RealisedProcess*. Possible changes include those to *Product*, *Engineer* and *Context*. For example, an *Activity* involving engineers coding may result in change to both *Product* and *Engineer* states and one involving a discussion between engineers and customers may result in change to only *Engineer* states.

Activity effects change by providing *ContextModel* with the transformations supplied by *Method* and *Technique* and applying the modified transformation delivered by *ContextModel*.

Note that, even when the same *Technique* is applied by the same *Engineer* in the same *Context*, the final states for *Product*, *Engineer* and *Context* may be different if a different *ContextModel* is applied.

Detailed description

Type Transformation.

Purpose Effect change to the state of the *RealisedProcess*.

Function *Activity* effects a single state change to *RealisedProcess*. The state change involves changes to one or more of *Product*, *Engineer* and *Context* states.

Subordinates None.

Dependencies *Product*, *Engineer* and *Context* ('transforms'); *ContextModel*, *Method* and *Technique* ('mapConstrainedBy').

Interface Interfaced to by *RealisedProcess* ('hasTransformation').

Processing An *Activity* transforms *Product* according to *Method*, *Technique* and *ContextModel*. *Activity* transforms *Engineer* and *Context* according to *ContextModel*.

Data Binary (functional) relation on *RealisedProcess* state representing possible start states (domain) and end states (range).

Formal description

Activity describes a functional mapping on *Product*, *Engineer* and *Context*. The set of all *Activity* is \mathcal{A} .

Transformation to *Product* results from adjusting the transformation defined by *Technique* according to *ContextModel*. The resulting transformation is applied to the *ProductMeasurements* for $ps \in PS^p$. Transformation to *Engineer* is according to $cmeng$.

The relation that describes *Activity* is

$$A : (\mathcal{PS} \times \mathcal{ES} \times \mathcal{CS}) \mapsto (\mathcal{PS} \times \mathcal{ES} \times \mathcal{CS}) \quad (7.38)$$

An *Activity* a is associated with a *Method* m , a *Technique* t and a *ContextModel* cm . From Equation 7.33, the transformation effected by m and t is $t \circ m$. For *Activity* a , I denote this by a_{tom} and represent by the relation (ps, ps'') . The *ContextModel* for a , denoted by a_{cm} , modifies this to (ps, ps') . The domain for a_{cm} is thus restricted to *Product* states for which a_{tom} is ps'' .

Applying Equations 7.34 and 7.38

$$a: (ps, es, cs) \mapsto (ps', es', cs') \mid \\ \exists ps'': a_{tom}(ps) = ps'' \wedge a_{cm}(ps, ps'', es, cs, t) = (ps', es', cs') \quad (7.39)$$

Activity has the properties that at least one of ps , es or cs must change.

$$\forall rp \in \mathcal{RP}, \forall a \in \mathcal{A} \quad a(ps, es, cs) = (ps', es', cs') \mid \\ (ps, ps' \in PS^{rp} \wedge es, es' \in ES^{rp} \wedge cs, cs' \in CS^{rp}) \wedge (ps \neq ps' \vee es \neq es' \vee cs \neq cs') \quad (7.40)$$

7.2.14 RealisedProcess

Overview

A *RealisedProcess* is described as a directed graph with nodes representing states of the *RealisedProcess* and edges representing transitions between these states. Events that cause change to the state of a *RealisedProcess* may be planned by management or unplanned. Examples of the former are ‘start working on an *Activity*’ and ‘current *Activity* has been completed’. Examples of the latter represent either a change to project *Context*, for example, ‘replace engineers at short notice’, or the need to ‘change the plan’ by interrupting an *Activity* that is partially complete. Because of the unplanned events, a *RealisedProcess* must be able to react to the environment and an element of ‘event-response’ is introduced i.e. we do not have a simple dataflow situation.

A *RealisedProcess* is made up of *Product*, *Engineer*, *Context*, some *Activities* and a *GoalsBenchmark*. *Product* represents what is delivered from the *RealisedProcess* and *GoalsBenchmark* provides a means of checking ‘readiness to deliver’. *Engineer* represents people who change *Product* and they do so by involvement in *Activities*. *Context* provides information that affects how well people work when involved in the *Activities*.

Detailed description

Type Aggregation.

Purpose Abstract the *software process* as it is realised i.e. descriptive.

Function Transforms the state of the software process from the state at project start to a new state at project end.

Subordinates *Activity* ('hasTransformation'); *Product*, *GoalsBenchmark*, *Engineer* and *Context* ('hasStates').

Dependencies None.

Interface None.

Processing A *RealisedProcess* applies *Activities* to transform *RealisedProcess* states. State transformation also occurs as a result of external perturbations, for example, when engineers are replaced or *Activities* interrupted in an unplanned way.

Data A directed graph that captures changes in the state of *Product*, *Engineer*, *Context* and *Activity*. Nodes represent states of the *RealisedProcess* and arcs represent state transitions. There is a single 'start' node, corresponding to the state of the *RealisedProcess* at project start and a single 'end' node, corresponding to its state at project end. A *GoalsBenchmark* captures states that are desirable as end states. The states for a *RealisedProcess* are constrained by the product, engineers and context involved in the process. For example, if the *RealisedProcess* acts on product 'MyWebApp version 2.1' and has engineers 'Joe', 'Mike' and 'Barbara', the state space is constrained to include possible states of this product and set of engineers.

Formal description

The set of all *RealisedProcess* is \mathcal{RP} . The *RealisedProcess* for the *Project* pr is rp .

I consider a state machine representation to model rp . The state space for our state machine for rp is $PS * ES * CS * A$. In order to 'mirror' the real-world as closely as possible, I capture points of visibility into the *RealisedProcess* as input stimuli, and select the following events:

startActivity(a) Start work on *Activity* a . *Product* is in a known state, engineers have been selected and context is known (i.e. PS , ES , CS and $a \in A$ are defined).

changeContext Some state change to *Engineer* or *Context* is to be applied.

endActivity Stop work on the current *Activity*.

A state machine representation for *RealisedProcess* is presented in Figure 7.5.

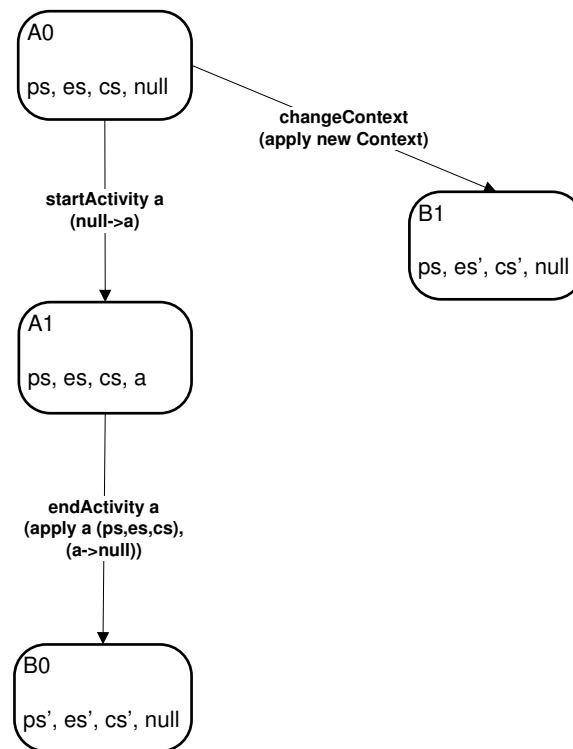


Figure 7.5: State machine for KiTe

Initial state *A0* has *Product*, *Engineer* and *Context* in states *ps*, *es* and *cs*, respectively. No *Activity* is active and so we have *Activity* ‘null’. On stimulus ‘start work on *Activity a*’, the ‘null’ *Activity* state becomes ‘a’ i.e. state *A1*. This state is applicable throughout application of *Activity a*. As we have no visibility into the process during this time, *Product*, *Engineer* and *Context* remain in states *ps*, *es* and *cs*. When *Activity a* completes, either because of planned completion or unplanned interruption, states *PS*, *ES* and *CS* transition to *ps'*, *es'* and *cs'* as a result of full or partial application of *Activity a*, and *a* returns to the ‘null’ state. Note that application of a *KiTe Activity* may result in change to one or more of *Product*, *Engineer* or *Context*. On stimulus ‘changeContext’, the *RealisedProcess* moves to state *B1*. In this state *es* or *cs* (or both) have changed, for example, engineers have been replaced.

In a real-life project, a ‘changeContext’ stimulus might occur when a *Activity* is active, for example, if an engineer is unexpectedly sick and must be replaced. This situation may in fact be represented by the input events already discussed. In Figure 7.6, I illustrate this situation. For a change in *Context* when a *Activity* is active, we must first stop work on *Activity* and transition to the new state according to completion status (i.e. ‘endActivity’), then apply a ‘changeContext’ transition, and finally a ‘startActivity’ where the *Activity* may be a modified version of the original.

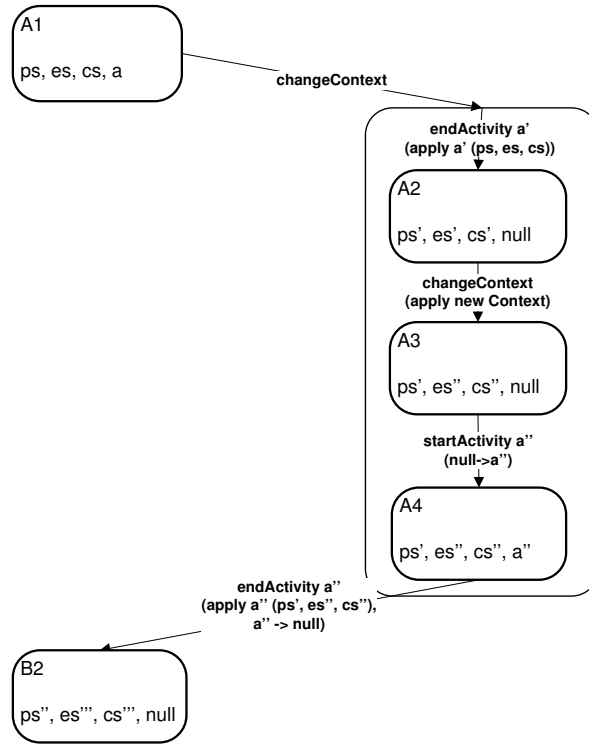


Figure 7.6: Disturbing a Method

The formal representation for a *finite automaton* is [62]

$(Q, \Sigma, \delta, q_0, F)$ where

Q is a finite set of states,

Σ is a finite *input alphabet*,

δ is the transition function mapping $Q \times \Sigma$ to Q ,

q_0 is the initial state,

$F \subseteq Q$ is the set of final or accepting states.

The representation for *RealisedProcess* is:

$rp = (Q, \Sigma, \delta, q_0, F)$ where

$Q = \{PS^{rp} \times ES^{rp} \times CS^{rp} \times A\}$,

$\Sigma \equiv \{\text{'StartActivity(a)'}, \text{'ChangeContext'}, \text{'EndActivity'}\} \mid a \in A$,

$\delta : Q \times \Sigma \mapsto Q$ is shown in Table 7.1,

q_0 = initial state for rp,

$F \subseteq Q = GB$.

Note that *GoalsBenchmark* represents a specific set of product states i.e. the ‘desired’ end states. These may be considered to be the ‘accepting states’ for the *RealisedProcess*. A *RealisedProcess* actual end state may or may not be one of the ‘accepting’ states, for example, if

Table 7.1: State transition diagram for RealisedProcess

	StartActivity(a)	ChangeContext	EndActivity
ps, es, cs, null	ps, es, cs, a	ps', es', cs', null	-
ps, es, cs, a	-	-	ps', es', cs', null

product-related objectives are not met.

8

Evidence

In Section 5.3.1, I proposed some objectives for *KiTe*. These were:

- Capture any software process or process model.
- Compare processes and process models.
- Create a new process by combining elements from different processes.

The conventional scientific approach towards evidence accumulation is to spawn hypotheses based on the theory or model of interest and carry out formal experiments that aim to disprove the theory. The idea of a ‘null hypothesis’ is central to this experimental paradigm. According to Dawson et. al., this represents a *positivist* approach i.e. where the researcher “looks for ir-refutable facts and fundamental laws that can be shown to be true regardless of the researcher and the occasion” [38]. He reminds us that software engineering is “not a pure science” and it is “arguable whether a positivist approach can ever be appropriate for a discipline so dependent upon people and the environment”. For this reason, many researchers favour an *interpretivist* approach i.e. where researchers interpret results within the context in which the research takes place. Such an approach can lead to “new, empirically grounded theories”, but not directly to the discovery of cause-and-effect relationships. Dawson et. al. cite the example of impressive results when a new methodology is applied being accredited to the methodology, when results could have been “due to something as simple as the higher motivation achieved by a pay

rise” [38]. The authors make a case for accumulating many little evidences of different kinds and reference the fields of medicine, law, industrial engineering and knowledge management to support their case. This approach compares with that of other researchers who draw from the fields of psychology and sociology to suggest the accumulation of an ‘evidence portfolio’ comprising case studies, anecdotes, surveys, expert opinion and controlled experiments [87, 160].

As a means of capturing *KiTe*’s ability to meet its objectives, I have chosen an approach called *argumentation* along with an established notation, *Goal Structure Notation (GSN)*, for structuring and capturing arguments. This approach has been used for many years in the safety critical domain and has recently been applied in the software domain [160].

Argumentation is “an approach which can be used for describing how evidence satisfies requirements and objectives” [160]. The use of a suitable notation such as *GSN* helps researchers to easily identify what evidence is required and helps stakeholders see at a glance what is the ‘evidence coverage’. I have chosen this strategy for two reasons. The first relates to the fact that the breadth of evidence required is large, and this approach helps organise and display evidence such that both evidence and lack of evidence are relatively easy to see. The second reason relates to the idea presented above that, for people-intensive systems, accumulating a portfolio of different kinds of evidence is appropriate. The argumentation approach provides for individual items of evidence and so is suitable for the portfolio idea.

My selection of studies to provide evidence has depended upon a mix of strategy and pragmatism. The strategy has been to aim to provide a breadth of evidence in the first instance, and so an attempt has been made to choose studies that maximise the number of evidence goals met in the time available. Pragmatism has focussed the choice of study to those with least time overhead i.e. studies recreate processes and models described in the literature. The use of an evidence map enables the reader to easily see what is the ‘evidence cover’ and gain some idea of the strength of the evidence at a glance. In the next Section, I introduce the *evidence map* and explain how it is used to show what evidence is available to support the *KiTe* objectives. In Sections 8.2 and 8.3, I present my evidence. Finally, I discuss some interesting points brought to light as a result of the evidence-gathering exercise.

8.1 Evidence map

According to Weaver et. al., there is a trend in modern safety critical standards away from prescriptive, process-based standards towards the use of a ‘safety case’ with supporting evidence [160]. Each safety case comprises three principal elements — *Goal*, *Argument* and *Evidence*. A high level goal may be presented as a *goal hierarchy* with supporting arguments and evidence attached to each sub-goal [47]. Weaver et. al. suggest that “Argument without sup-

porting evidence is unfounded, and therefore unconvincing” and “Evidence without argument is unexplained — it can be unclear that . . . objectives have been satisfied”. This approach is called *argumentation* and its purpose is to communicate clearly, comprehensively and defensibly that a system meets its safety goals [160].

Goal Structure Notation (GSN), is a graphical argumentation notation where goals are presented as rectangles, arguments as parallelograms and evidence as circles. The goal hierarchy is thus a structure showing how goals are addressed by arguments and how arguments are supported by evidence. Arguments for which no evidence exists, i.e. undeveloped goals, have an attached diamond.

I apply this approach to *KiTe* by stating the objectives defined in Section 5.3.1 as a goal hierarchy and providing arguments and evidence for each goal in the hierarchy. For example, the main objective for *KiTe* is “Provide evidence for *KiTe*” and, in Figure 5.1, I show the top level objectives as ‘Capture’, ‘Compare’ and ‘Combine’. In Figure 8.1, I place the main objective as the root in the goal hierarchy and provide three arguments, each with evidence realised as a sub-goal. For example, the first argument is “Argument by showing any software process can be captured in *KiTe*” and the ‘evidence’ sub-goal is “Capture any software process or process model”. Sub-goals 2 and 3 correspond to the remaining *KiTe* objectives. The remaining structure is built according to the objectives defined in Figures 5.1. ‘Leaf’ goals either have some available evidence (shown as attached circles containing the number of a case study) or no available evidence (shown as an attached diamond). Each ‘evidence circle’ references a study that contains information about how the goal has been met and these are summarised in the legend and presented in detail in the following Sections.

In Figure 8.1, I have not provided arguments and evidence for the sub-goals 1.2 (“Capture software development processes and process models”) and 1.4 (“Capture miscellaneous processes”). These sub-goals are realised in Figures 8.2 and 8.3.

Note that this ‘evidence map’ captures breadth of evidence rather than depth i.e. the strength of individual pieces of evidence must be obtained from studying the actual evidence. The strength of the map is rather to provide a quick idea of what evidence exists and easy access to that evidence [160].

8.2 Capture all Processes and Process Models

In this Section, I present the studies that comprise evidence to support the objective of ‘Capture any software process or process model’ (Goal 1). For each study, I describe my reasons for selecting the particular study in terms of which goals are satisfied by the study.

As I represent various processes and models in *KiTe*, I notice that, in all cases, the attempt

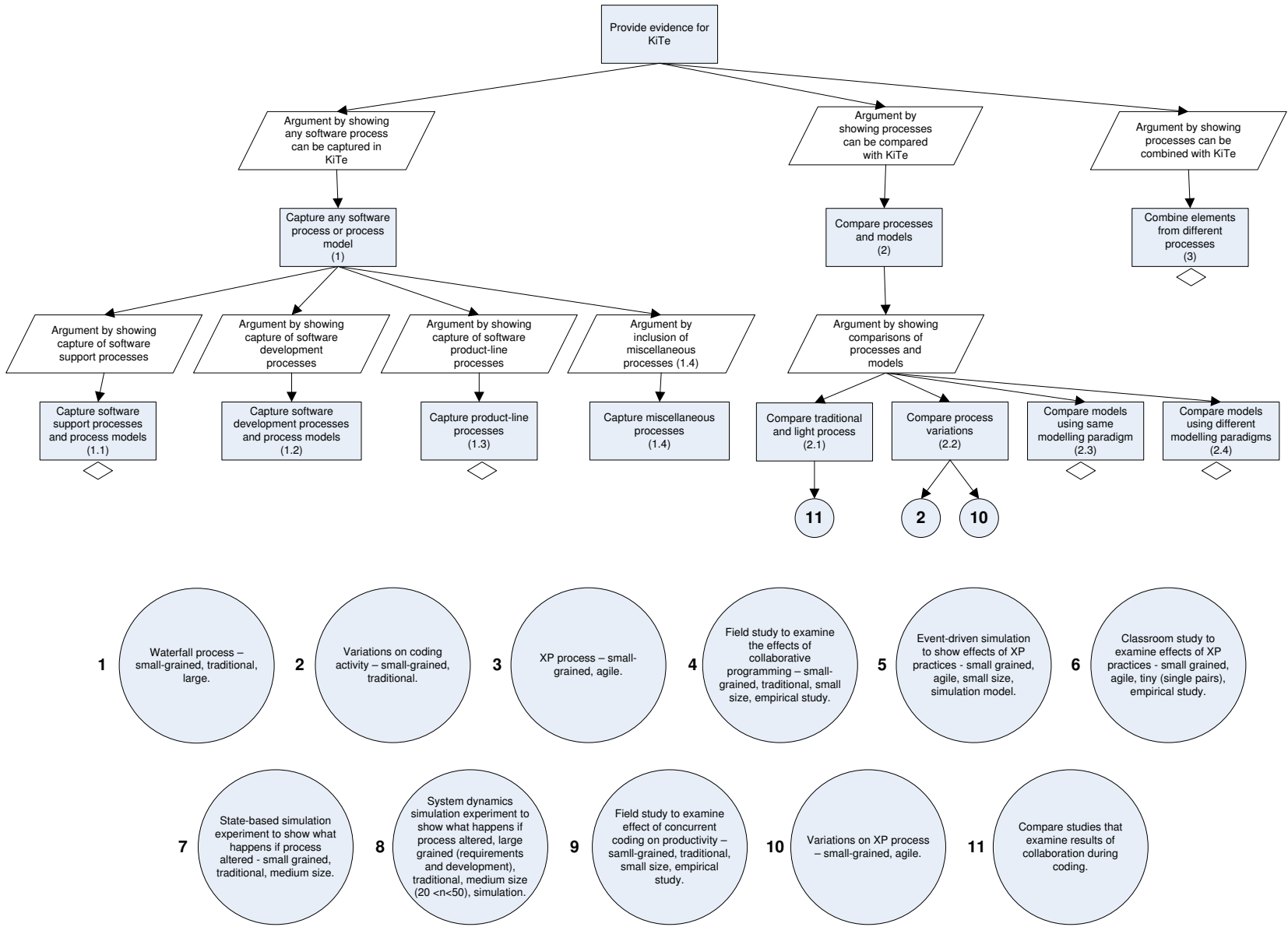


Figure 8.1 : KiTe goal hierarchy

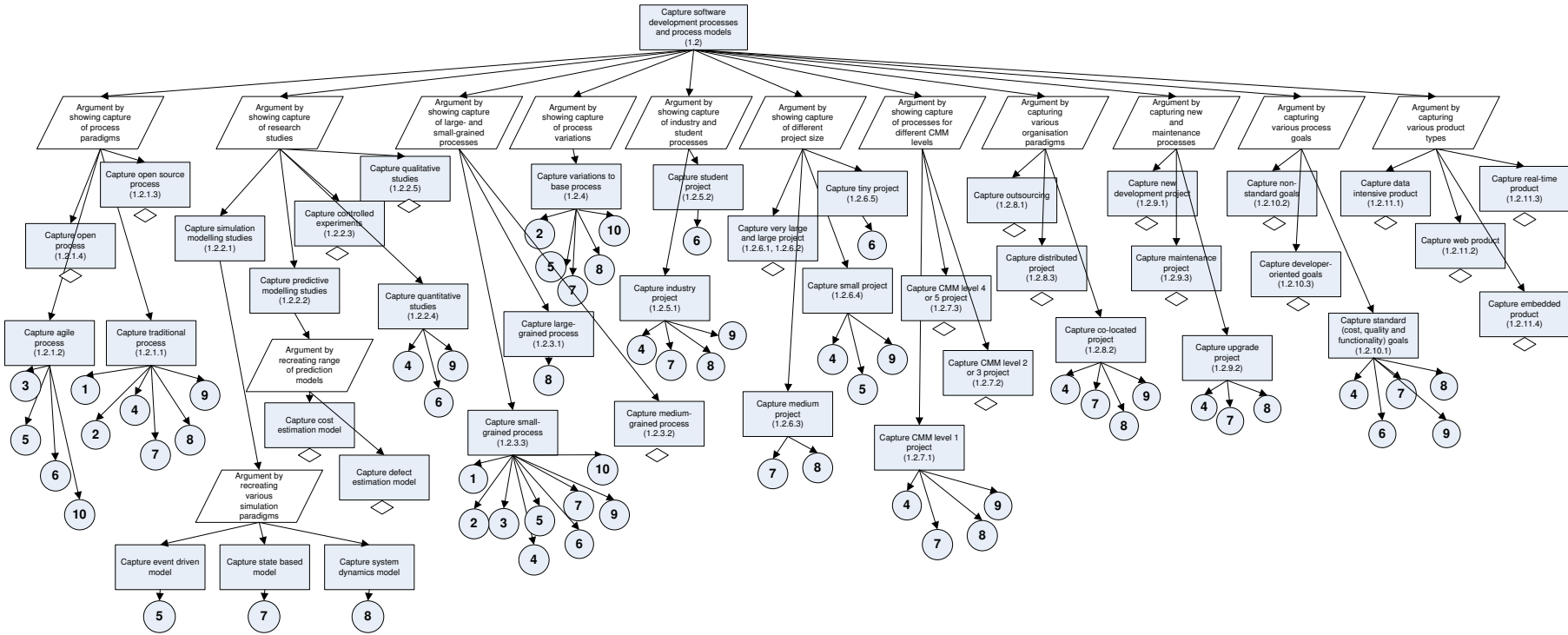


Figure 8.2: Goal hierarchy for 'Capture'

- 1 Waterfall process – small-grained, traditional, large.
- 2 Variations on coding activity – small-grained, traditional.
- 3 XP process – small-grained, agile.
- 4 Field study to examine the effects of collaborative programming – small-grained, traditional, small size, empirical study.
- 5 Event-driven simulation to examine effects of XP practices - small grained, agile, small size, simulation model.
- 6 Classroom study to examine effects of XP practices - small grained, agile, tiny (single pairs), empirical study.
- 7 State-based simulation experiment to show what happens if process altered - small grained, traditional, medium size.
- 8 System dynamics simulation experiments to show what happens if process altered, large grained (requirements and development), traditional, medium size (20 <n<50), simulation.
- 9 Field study to examine effect of concurrent coding on productivity – small-grained, traditional, small size, empirical study.
- 10 Variations on XP process – small-grained, agile.
- 11 Compare studies that examine results of collaboration during coding.

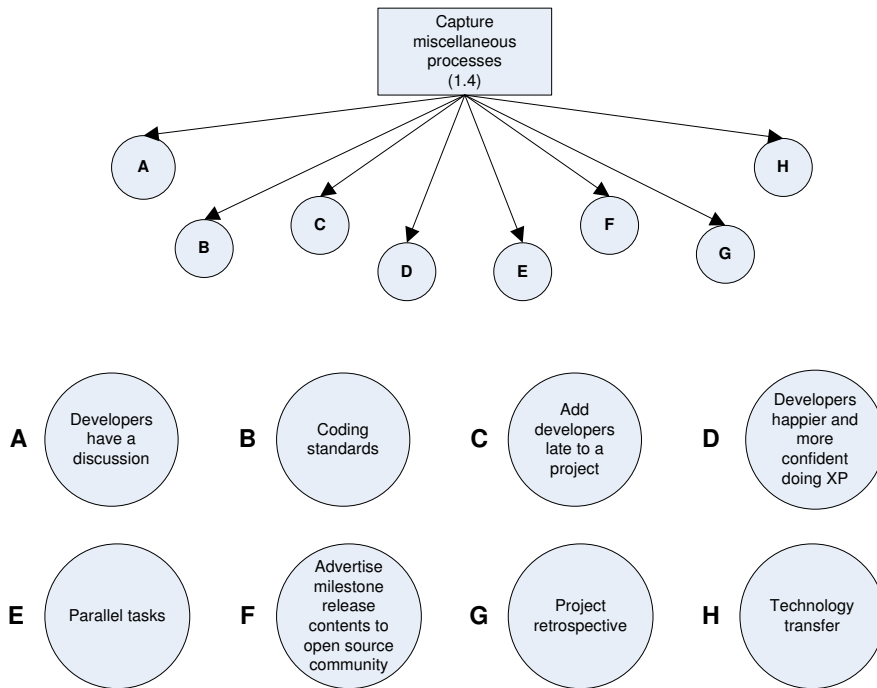


Figure 8.3: Goal hierarchy for 'Miscellaneous'

results in exposure of ambiguities or unstated assumptions. There could be many reasons why these ambiguities and assumptions are present, such as space constraints resulting in authors having to leave out some details. However, this does not detract from the fact that the attempt to represent a study in a framework such as *KiTe* provides an excellent means of exposing lack of clarity. This is an important side-effect of the use of a model such as *KiTe*. When ambiguities and assumptions are brought to light, I continue as if the study had been more fully documented by making a choice and 'fixing' the uncertain aspects. I believe the value of the evidence is not affected by this, as in most cases it is clear that the actual choice made is not important when considering the ability of *KiTe* to represent. Specific examples of process variations are also included as evidence studies (see Sections 8.2.2 and 8.2.10) and this provides some confidence that alternative descriptions may be easily represented.

In order to capture a process or process model in *KiTe*, it is necessary to:

1. Capture the attributes of interest for *Product* in a model of *Product Perspectives*. This may include *Content*, *Quality* and *Cost Perspectives* and any other *Perspectives* relevant to the modelling exercise.
2. Define the *Methods* to be performed by stating the precondition for each and how it transforms *Product*. Specify the *Technique* to be implemented.
3. Define *Engineer* by capturing capabilities in an *Engineer CapabilitySpec* i.e. define *ES*.

4. Define *Context* factors that might affect how well the *Methods* are carried out i.e. define *CS*.
5. Define *ContextModel* i.e. the mechanism for transforming *Product*, *Engineer* and *Context*. This involves defining the capabilities required for working with *Technique* and *Product* i.e. *Technique CapabilitySpec* and *Product CapabilitySpec* along with the rules for transforming *PS*, *ES* and *CS*.
6. Define the ‘acceptance criteria’ for the project i.e. the product-related objectives. These are attributes of *Product* that have agreed values at the end of the project, for example, ‘90 percent of the requirements implemented and fewer than 20 defects’ or ‘development cost less than twenty thousand dollars’. In *KiTe*, this is our *GoalsBenchmark*, *GS*.

In this Section, I follow the above steps. Note that, for those studies in which I capture a ‘general’ process, for example, “capture a waterfall process”, the attributes applied for *Product*, *Engineer* and *Context* are selected to best represent the process as reported in the literature. I then discuss this in the light of the *KiTe* framework. For example, for waterfall-based studies, the tendency is to report product-related attributes only, for example size, defect and cost, whereas, in an XP study, reporting tends to include engineer-related attributes. For such studies, I also choose attribute values that are representative only i.e. as the study is not based on an actual experiment, I choose values that seem to capture the ‘flavour’ of the process as reported in the literature. For the reasons discussed earlier, the issue of measurement scales is out of scope for this thesis and I manipulate values in an informal way and without discussing such issues.

8.2.1 Study 1: Waterfall process

For my first study, I select a simple, well-known process model that is implemented in many projects — the Waterfall. This is a traditional process (Goal 1.2.1.1.) and I represent at a small level of granularity (Goal 1.2.3.2.).

Because this study aims to represent a ‘typical’ waterfall process, I choose attributes for *Product* that are representative of those found in the literature, and comment that, in order to represent with different attributes, I need simply change the model for *Product* to include the required attributes. For the same reason, I work with specific attribute values. For example, the model for *Product* has an attribute ‘number of requirements’ and for my illustration I choose ‘30’ as a value.

For a ‘pure’ waterfall process, i.e. one that adheres to a manufacturing process, each stage is carried out only once. The analogy with manufacturing breaks down at this point as the software equivalent to ‘throwing away defective items’ is generally to fix some of the defects prior to

delivery and some iteration to earlier stages is thus inherent in the model. In a more realistic process, iteration may occur in a number of different ways. For this study, I will illustrate both the simple model and a model with a single iteration, and discuss other possibilities. I also assume all waterfall *Methods* complete. The ability to represent situations where a *Method* is disturbed for some reason is illustrated in Section 8.2.11.

KiTe representation

The first task is to define an appropriate *Product* model. Models described in the literature generally contain some measure of product size, for example, ‘lines of code’, ‘number of requirements’ or ‘function points’, some metric involving defect numbers and some measure of cost, for example, ‘person weeks’ or ‘duration’. In a waterfall process, generally all *Partitions* are affected. For this illustration I will apply the *Product* model in Table 8.1.

Table 8.1: Waterfall Product Model

Perspective	Partition	Attribute	Meaning	
Content	Definition	# Requirements	# requirements captured	
	Architecture	# Requirements	# requirements architected	
	Design	# Requirements	# requirements designed	
	Source	# Requirements	# requirements implemented	
	Integration	# Requirements	# requirements integrated	
	Packaged	# Requirements	# requirements packaged	
	Quality	Definition	RemainingDefects	# remaining defects in requirements
		KnownDefects	# known defects in requirements	
Architecture		RemainingDefects	# remaining defects in architectures	
		KnownDefects	# known defects in architectures	
Design		RemainingDefects	# remaining defects in designs	
		KnownDefects	# known defects in designs	
Source		RemainingDefects	# remaining defects in source	
		KnownDefects	# known defects in source	
Integration		RemainingDefects	# remaining defects in integrated source	
		KnownDefects	# known defects in integrated source	
Packaged		RemainingDefects	# remaining defects in packaged source	
		KnownDefects	# known defects in packaged source	
Cost		Definition	DurationWeeks	Duration in weeks for requirements
		Architecture	DurationWeeks	Duration in weeks for architectures
	Design	DurationWeeks	Duration in weeks for designs	
	Source	DurationWeeks	Duration in weeks for source	
	Integration	DurationWeeks	Duration in weeks for integration	
	Packaged	DurationWeeks	Duration in weeks for packaging	

I next identify the *Methods* that are applied to transform the *Product*. A waterfall model generally is described as a number of stages, with each stage resulting in the production of different kinds of product document (see Section 2.2). For example, a ‘design’ stage results in the production of design documents. I select a *Method* for each stage, and apply *Methods* ‘CaptureRequirements’, ‘Analyse’, ‘Design’, ‘Code’, ‘Integrate’ and ‘Test’. ‘CaptureRequirements’ results in change to the *Definition Partition* for all *Perspectives*. ‘Analyse’, ‘Design’ and ‘Code’ affect the *Architecture*, *Design* and *Source Partitions* respectively. ‘Integrate’ causes change to both *Integration* and *Packaged Partitions* and ‘Test’ results in change to all *Partitions*, because some defects are brought to light and are now known and perhaps some of these defects are resolved in the relevant *Partitions*. I note that my choice of *Method* granularity may help us to understand some aspects of the waterfall process, but may not help with other aspects. However, the choice is suitable as a first step and sufficiently simple to aid illustration.

The next task is to identify models for *Engineer* and *Context* along with the *ContextModel* that defines the effects of the human-related factors on the process. Because the waterfall paradigm is a manufacturing one and waterfall projects are traditionally lengthy, human factors are generally not captured i.e. the waterfall mindset is that the effects of such factors are negligible or ‘average out’ over the duration of a project. However, the degree of tool support for the various *Methods* and *Techniques* is generally believed to be of relevance and so I include a *Context* attribute ‘ToolSupport’ applied to each *Partition*. An illustrative *Context* model is shown in Table 8.2. *ContextModel* uses values for ‘ToolSupport’ to calculate engineer effectiveness when working with *Techniques*. *ContextModel* has no effect on *Engineer* or *Context*.

Table 8.2: Waterfall Context Model

Partition	Attribute	Meaning
Definition	ToolSupport	level of tool support for requirements gathering
Architecture	ToolSupport	level of tool support for architecture
Design	ToolSupport	level of tool support for design
Source	ToolSupport	level of tool support for coding
Integration	ToolSupport	level of tool support for integration
Packaged	ToolSupport	level of tool support for packaging

In Figure 8.4, I illustrate a *RealisedProcess* based on a single-pass waterfall model. The *Methods* identified above form the basis of *Activities* ‘Requirements’, ‘Analysis’, ‘Design’, ‘Coding’, ‘Integration’ and ‘Testing’. I illustrate the effects of the *RealisedProcess* on *Product* only, as the selected *ContextModel* implies no change to *Engineer* or *Context*. At *process* start, no work has been carried out on *Product* and the value for all attributes is ‘0’. After the

‘Requirements’ *Activity*, 30 requirements have been defined, 5 requirements defects have been injected and 8 weeks have elapsed. After the ‘Analyse’ *Activity*, all defined requirements have been architected and an additional 7 defects have been injected into the *Architecture* documentation. The ‘Design’ *Activity* causes injection of 16 design defects and ‘Coding’ results in 44 additional defects in the *Source*, making a total of 72 defects. The ‘Integration’ *Activity* carries the 30 requirements and all defects through into *Integration* and *Packaged* artifacts and no new defects are injected. After the ‘Testing’ *Activity*, 50 of the 72 defects have become visible, 4 sourced in the *Definitions*, 5 in the *Architectures*, 12 in the *Designs* and 29 in the *Sources*.

	Product Defn			Arch			Design			Source			Integrn			Packaged				
	# Requirements	Remaining Defects	Known Defects	DurationWeeks	# Requirements	Remaining Defects	Known Defects	DurationWeeks	# Requirements	Remaining Defects	Known Defects	DurationWeeks	# Requirements	Remaining Defects	Known Defects	DurationWeeks	# Requirements	Remaining Defects	Known Defects	DurationWeeks
Start	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Requirements	30	5	0	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Analysis	30	5	0	9	30	12	0	20	0	0	0	0	0	0	0	0	0	0	0	0
Design	30	5	0	9	30	12	0	21	30	28	0	10	0	0	0	0	0	0	0	0
Coding	30	5	0	9	30	12	0	21	30	28	0	11	30	72	0	10	0	0	0	0
Integration	30	5	0	9	30	12	0	21	30	28	0	11	30	72	0	12	30	72	0	1
Testing	30	5	4	9	30	12	9	21	30	28	21	11	30	72	50	12	30	72	50	1

Figure 8.4: Simple waterfall

As noted above, in practice at least one iteration occurs. I illustrate in Figure 8.5 a simple case in which a single iteration is applied to correct defects. As there are requirements defects, the iteration starts from the beginning and an *Activity* for resolving defects is carried out for each *Partition*, in turn. The identified defects are corrected at each stage, while durations for each *Partition* increase.

If I now assume that the expectation is that “for delivery, it is expected that at least 29 of the requirements are implemented and the number of defects is less than 30”, I see that the illustrated *RealisedProcess* has achieved its goals.

Recall that, in *KiT*e, a *RealisedProcess* rp is represented as (see Section 7.2)

$$rp = (Q, \Sigma, \delta, q_0, F) \text{ where}$$

$$Q = \{PS \times ES \times CS \times A\},$$

$$\Sigma = \{‘StartActivity(a)’, ‘ChangeContext’, ‘EndActivity’\} \mid a \in A,$$

$$\delta : Q \times \Sigma \mapsto Q \text{ is shown in Table 7.1,}$$

	Product Defn			Arch			Design			Source			Integrn			Packaged				
	# Requirements	Remaining Defects	Known Defects	DurationWeeks	# Requirements	Remaining Defects	Known Defects	DurationWeeks	# Requirements	Remaining Defects	Known Defects	DurationWeeks	# Requirements	Remaining Defects	Known Defects	DurationWeeks	# Requirements	Remaining Defects	Known Defects	DurationWeeks
Start	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Requirements	30	5	0	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Analysis	30	5	0	9	30	12	0	20	0	0	0	0	0	0	0	0	0	0	0	0
Design	30	5	0	9	30	12	0	21	30	28	0	10	0	0	0	0	0	0	0	0
Coding	30	5	0	9	30	12	0	21	30	28	0	11	30	72	0	10	0	0	0	0
Integration	30	5	0	9	30	12	0	21	30	28	0	11	30	72	0	12	30	72	0	1
Testing	30	5	4	9	30	12	9	21	30	28	21	11	30	72	50	12	30	72	50	1
RequirementsResolve	30	1	0	11	30	12	9	21	30	28	21	11	30	72	50	12	30	72	50	1
AnalysisResolve	30	1	0	11	30	3	0	23	30	28	21	11	30	72	50	12	30	72	50	1
DesignResolve	30	1	0	11	30	3	0	23	30	7	0	13	30	72	50	12	30	72	50	1
CodingResolve	30	1	0	11	30	3	0	23	30	7	0	13	30	22	0	14	30	72	50	1
Integration	30	1	0	11	30	3	0	23	30	7	0	13	30	22	0	14	30	22	0	2
Testing	30	1	0	11	30	3	0	23	30	7	0	13	30	22	2	14	30	22	2	2

Figure 8.5: Waterfall with defect resolution

$q_0 =$ initial state for rp ,

$$F \subset Q = GB.$$

For the waterfall examples illustrated above, ES and CS contain a single state i.e. any change to *Engineer* or *Context* is not captured. GB is the set of ‘accepting’ states i.e. all product states in which the value of the ‘Content: Packaged: # Requirements’ attribute is 29 or greater and the value of the ‘Quality: Packaged: RemainingDefects’ attribute is smaller than 30. *Product* is defined by Table 8.1 and *Context* by Table 8.2.

Each row in Figure 8.5 represents the state of *Product* on completion of an *Activity*. PS is the conjunction of this set of states and GB . q_0 is the set of values in the first row of the figure, i.e. the row labelled ‘Start’. Note that, in an enhancement or maintenance project, values in the ‘Start’ row would be other than 0, i.e. there would be some *Product Content*.

The *Activities* shown in Figure 8.5 are associated with the set of *Methods*

$$M = \{ \text{‘CaptureRequirements’, ‘Analyse’, ‘Design’, ‘Code’, ‘Integrate’, ‘Test’, ‘ResolveRequirementDefects’, ‘ResolveArchitectureDefects’, ‘ResolveDesignDefects’, ‘ResolveCodeDefects’} \}.$$

In the example, the ‘acceptance criteria’ are met i.e. the *RealisedProcess* reaches an ‘accepting state’.

Discussion

Several points for discussion emerge from the above study. The first involves the granularity of the *Methods* selected. I chose a single *Method* for each phase, for example, ‘Analyse’. However, as each waterfall phase involves testing of the produced artifacts, I could have chosen to replace the single *Method* with both an ‘Analyse’ and ‘Review’ *Method*. In this case, I would have had a deeper insight into the existence of ‘KnownDefects’, as these would have become apparent as a result of ‘Review’ and the information could have been used to initiate an ‘Analyse’ iteration to correct defects and prevent their propagation to designs. I chose the more simple process because I wanted to avoid decisions about iteration that would confuse the illustration.

The second point follows from the first and relates to the consideration of what exactly is meant by ‘waterfall’ model. At each phase, a number of defects are injected. Reviews expose some of these defects, along with the *Partitions* in which they are sourced. If some discovered defects are sourced in an ‘earlier’ *Partition*, for example, defects found in designs but sourced in requirements, some decision has to be made about whether or not to iterate back to requirements immediately. There are possibly many such decision situations in a waterfall process. At one end of the range of possibilities is an iteration back to the earliest source for defects at each stage. At the other end is an iteration strategy that involves waiting until test before iterating. The situation is made more complex by the possibilities for test options at each stage. The possibilities range from no reviews (so there are no ‘KnownDefects’ and defects are propagated), to reviews with partial ‘local’ iteration i.e some defects are fixed and a decision must be made as to whether remaining ‘KnownDefects’ are propagated, to reviews with immediate and complete iteration (all ‘KnownDefects’ resolved).

The point to be made is that the label ‘waterfall’ is given to many different possible processes. As long as the actual policies are not visible, we have a situation fraught with ambiguity. Indeed, it is possible that the huge variation in outcomes reported for waterfall processes is, at least in part, due to the lack of definition of what is actually being done. The attempt to identify what are the *KiTe Methods* for a waterfall process bring to light these kinds of ambiguities, as it becomes clear that the specifics for *Method* are often unknown or unreported. I further illustrate this common lack of clarity in the next study.

In addition to the ambiguities uncovered above, I notice that *Engineer* and *Context* are modelled as unchanging during a waterfall process. This represents a source of assumptions. For example, in this model, if an engineer becomes sick and is replaced, the replacement engineer is indistinguishable from the original one. In terms of the model, the ‘ChangeContext’ input now has no effect on the *RealisedProcess* state i.e. represents a move to the same state. In terms of the real world, changes that may affect process outcomes are not modelled and it is

thus assumed that they do not affect process outcomes.

A final observation on the above study is that, if any of the details presented had been different, a different *RealisedProcess* would have been represented. For example, if the first *Activity* ‘Requirements’ had resulted in a ‘Remaining Defects’ value of 7, rather than 5, we have a different *Activity*, even if the implemented *Method* and *Technique* are the same in both cases. The reason is that some contextual factor has changed, for example, developers are less experienced, and so the *RealisedProcess* has changed also.

8.2.2 Study 2: Coding variations

This study also represents a waterfall model, but the purpose this time is to show how in *KiTe* I can represent some of the many possible variations that give rise to ambiguities when a waterfall is discussed. The study relates to a traditional (Goal 1.2.1.1.), small-grained (Goal 1.2.3.3.) process that captures variations to a base process (Goal 1.2.4.) and that compares process variations (Goal 2.2).

In this study, I consider the apparently straightforward problem of defining the *Method* ‘create code’. This description is commonly used to mean a number of things. For example, perhaps detailed designs are available from which to base code or perhaps the designs are incomplete and ‘experts’ are available to aid understanding and clarify uncertainties. Perhaps it is expected that a specific technique will be carried out, for example, pair programming.

Each of the above has different preconditions on *Product* and changes *Product* in different ways. In *KiTe*, they describe different *Methods*. I give an example of how some of the variations might be captured in *KiTe*.

KiTe representation

I first define a model for *Product*. I will work with a *Content Perspective* with attribute ‘number of requirements’ in each *Partition* and a *Quality Perspective* with ‘number of remaining defects’, as this model is sufficient to show capture and uncover issues. Rather than implementing a table to show the results of *Activities* on *Product*, as in the previous study, I present results as diagrams, as one of the aims of this study is to bring to light differences in meaning between processes commonly given the same name. As discussed in Section 8.2, I use attribute values in a ‘casual’ but illustrative manner.

For a traditional waterfall process, before coding takes place *Product* might look like the first graph in Figure 8.6. The x-axis presents the *KiTe Partitions* and the y-axis in an integer scale to represent ‘number of requirements’ and ‘number of defects’. Each graph contains two sets of columns — the left-hand (blue) column depicts ‘number of requirements’ for each *Partition* and the right-hand (green) column ‘number of remaining defects’. *Definitions, Architectures*

and *Designs* have a number of requirements implemented and each has some defects. There is no *Content* yet for *Source*, *Integration* and *Packaged*. I note that full *Designs* are in place and so coding may commence. The second graph in Figure 8.6 shows the state of *Product* after the coding task has completed. Now I have *Source* for all requirements and a number of *Source* defects. *Definitions*, *Architectures* and *Designs* are unchanged.

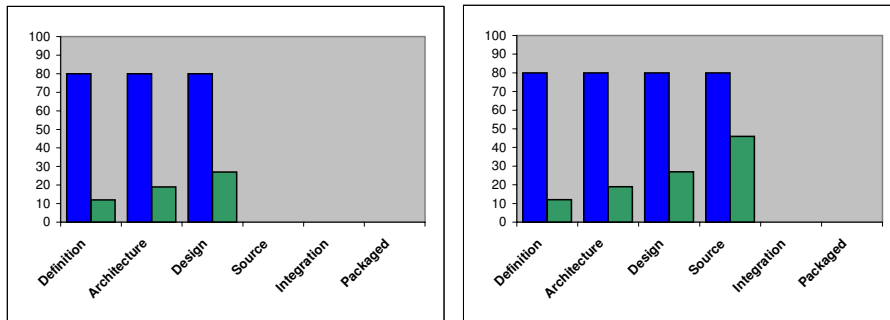


Figure 8.6: Coding in a waterfall process

I now want to capture what is arguably a more realistic version of events by considering the case where coding commences before designs are complete. The ‘before’ situation in *KiTe* is given by the first graph in Figure 8.7, where there are incomplete *Architecture* and *Design Content*. The ‘after’ graph again shows *Source Content* and no change to *Architecture* or *Design*. Because the precondition on *Product* is different in each of the two examples in Figures 8.6 and 8.7, in a *KiTe* model these are different *Methods*. A possible third scenario is one in which the ‘coding’ task expands to a ‘code and fix designs’ task. In this case, both *Design* and *Source Content* will change, and possibly also defect levels (Figure 8.8). Yet another *Method* is described.

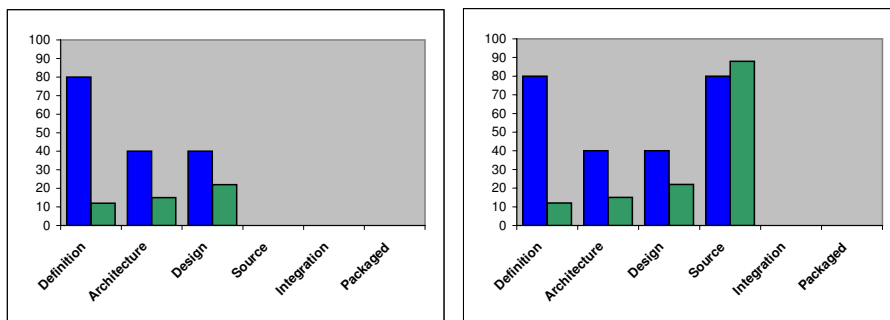


Figure 8.7: Coding before designs completed

For ‘coding’ in an XP-type process, I might have the graphs shown in Figure 8.9. *Content* is ‘number of user stories’ and in this example the ‘before’ graph shows that some user stories have been fully implemented (*Source*, *Integration* and *Packaged* all have *Content*) and a small number of new stories are ready for implementation. The coding process used includes building

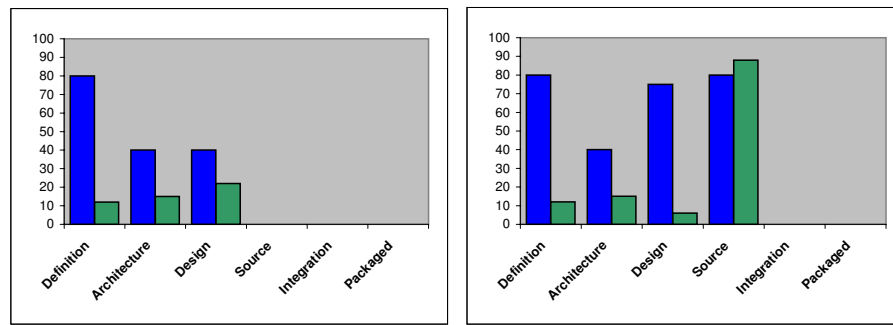


Figure 8.8: Coding and fixing designs

and packaging the product as is seen by the increased *Source*, *Integration* and *Packaged Content*, and then throwing away the implemented stories, as can be seen by the reduced *Definition Content* after coding. (I am aware that the stories would possibly be retained until after another process step involving customer acceptance, but I have taken some licence to aid description).

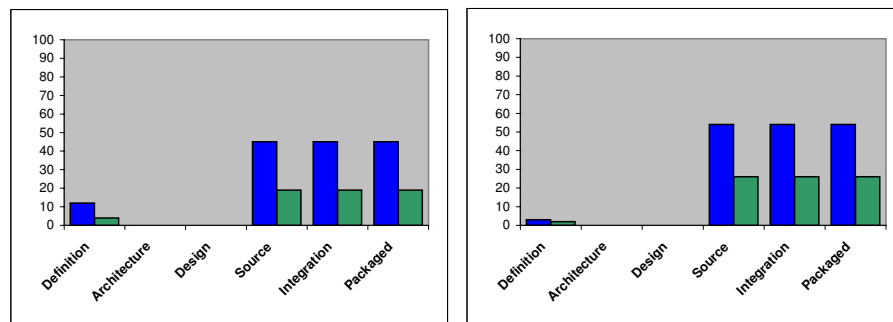


Figure 8.9: Coding in an XP process

For the examples illustrated above, our main focus is in the *KiTe Method*. *Method* is defined by its *Product* preconditions and effects on *Product*. The four examples above differ in either precondition or effect and thus represent different *Methods*.

Discussion

For simplicity, I did not show in the examples two other aspects of *KiTe*, those relating to *Context* and *Engineer*. In *KiTe*, the size of change in the ‘after’ pictures would depend upon the various process contexts applicable at the time of ‘coding’ and the *ContextModel* used for their capture. For example defect injection might be smaller if engineers were highly skilled and motivated or if some good tools were in place. Participation in a process changes engineers in *KiTe* and so missing from the diagrams is a model depicting *Engineer* attributes that change between ‘before’ and ‘after’ states.

This study illustrates the importance of being very specific when describing what is being done. Again, the attempt to capture ‘coding’ as a *KiTe Method* uncovers ambiguities that

effectively render comparison and prediction impossible.

8.2.3 Study 3: XP process

In the previous two Sections, I represented processes based on a traditional waterfall model. As much of the discussion in the literature concerns attempts to understand what are the differences between traditional and agile models, I now illustrate how a popular agile process, *XP (eXtreme Programming)* is represented in *KiTe* [85].

As for study 8.2.1, I aim to represent a ‘typical’ process and so choose attributes and values that are representative of those found in the literature. I use reported XP ‘evidence’ as a basis for decisions without questioning the soundness of this evidence. I also deal with attributes and measures as frequently found in the literature and do not attempt to justify the use of either. Although careful capture of attribute meaning is critical for empirical experimentation [89], the state-of-the-art remains immature in this area and consideration of associated issues is outside the scope of this study. For the same reasons, and for ease of illustration, I apply a ten-point scale as measures for those attributes that are based on a subjective evaluation. Also for ease of illustration, and because developers work interchangeably in an XP project, I treat all engineers as having identical *CapabilitySpecs*, and show only a single representative *CapabilitySpec*. For this study, I apply *Product* and *Engineer* transformations whose size is subjective. This means that the results of *Techniques* modified by *ContextModel* are based on a subjective representation of the literature. Again, the reasons are that the industry does not yet have ‘ideal’ models or sound experimental evidence, and the aim is, in any case, to illustrate concepts.

This study concerns an agile, small-grained process (Goals 1.2.1.2. and 1.2.3.3.).

KiTe representation

In order to capture an XP process [15] in *KiTe*, I examine each XP Practice and infer from it:

- Which characteristics of *Product* are changed by the Practice. These are included in appropriate *Product Perspectives*.
- Which characteristics of *Engineer* are affected by the Practice. These become *Engineer* attributes.
- Which characteristics of *Context* are affected by the Practice. These become *Context* attributes.
- How does the Practice relate to *Product* transformation? Does it describe a *Method*, a *Technique* or a constraint on the overall process?

In this way, I build up the *KiTe* models that are appropriate for XP.

Table 8.3 illustrates the result of identifying *Activities*, *Methods* and *Techniques*. I use names for these that help illustrate the issues when capturing a process in *KiTe*, and so some names are rather long, but hopefully helpful. Tables 8.4 and 8.5 summarise the required *Product*, *Engineer* and *Context* attributes. Selected attributes are inferred from claims in the XP literature, and aim to address perceived benefits. For example, for *Product*, I include ‘infrastructure’ because one of the mandates of XP is to create only what is required for current Stories with no additional structure to support future requirements. I also include ‘complexity’ because many believe that a lack of up-front design often results in overly complex code and because one of the XP Practices, ‘Refactoring’, is concerned with reducing code complexity. I use both terms without defining their meaning, in keeping with the use of the terms in the XP literature. For *Engineer*, I include *Engineer* ‘satisfaction’ and ‘confidence’ because it is claimed that participation in an XP project enhances these. I include an informal statement only of the meaning of each attribute.

Table 8.3: XP Activities

Activity	Method	Techniques
Planning	PlanningGame	SmallReleases, Metaphor
PairProgramming	DesignCodeAndUnitTest	PairProgramming, SimpleDesign, Metaphor, Refactoring, CollectiveOwnership, CodingStandards, OnSite-Customer, TestDrivenDesign
Integration	BuildAndUnitTestAndFixProblems	DeveloperBuilds, ImmediateProblemFix, Integrate-ToPackaged
CustomerTest	FunctionalTesting	

I now walk through a single cycle of an XP process in *KiTe* to illustrate how the various Practices affect *KiTe* models. I assume that a cycle has already been completed, and that engineers have reasonable technical skills and are sufficiently familiar with the subject area to start out with high confidence. Their knowledge of the product is small. Variations on this are presented in Section 8.2.10.

The situation at the start of the study is informally depicted in the first row of Figure 8.10.

A cycle has already been completed, and we have three Stories defined (*Content Definition* ‘Stories’) and two of these implemented (*Content Source/Implementation/Packaged* ‘Stories’). Developers are reasonably skilled (*Engineers* ‘TechnicalSkills’ 5) and comfortable with the subject area (*Engineers* ‘SubjectAreaKnowledge’ 5), but are unfamiliar with the product (*Engineers* ‘ProductKnowledge’ 1). As a result of this unfamiliarity, after the first cycle some additional

Table 8.4: XP Product Model

Perspective	Partition	Attribute	Meaning	
Content	Definition	Stories	# Stories captured	
	Source	Stories	# Stories implemented	
		Infrastructure	Code infrastructure (1-10)	
		UnitTests	# Stories with unit tests	
	Integration	Stories	# Stories integrated	
		Infrastructure	Integrated code infrastructure (1-10)	
		UnitTests	# Stories with integrated unit tests	
	Packaged	Stories	# Stories delivered	
		Infrastructure	Delivered code infrastructure (1-10).	
		UserTests	# Stories with functional tests	
	Quality	Definition	RemainingDefects	# remaining defects in captured Stories
			KnownDefects	# known defects in captured Stories.
Source		RemainingDefects	# remaining defects in implemented Stories	
		KnownDefects	# known defects in implemented Stories	
		Complexity	Code complexity (1-10)	
Integration		RemainingDefects	# remaining defects in integrated Stories	
		KnownDefects	# known defects in integrated Stories	
		Complexity	Code complexity (1-10) in integrated code	
Packaged		RemainingDefects	# remaining defects in delivered Stories	
		KnownDefects	# known defects in delivered Stories	
		Complexity	Code complexity in delivered code (1-10)	
Cost		Definition	PersonHours	Total person hours for dev group only

code infrastructure has been implemented (*Content Source/Implementation/Packaged* ‘Infrastructure’ 3) and the code exhibits some complexity (*Quality Source/Implementation/Packaged* ‘Complexity’ 3). Customer testing has found some implementation defects (*Quality Source/Implementation/Packaged* ‘KnownDefects’ 2) and also uncovered some changes required to Stories (*Quality Definition* ‘KnownDefects’ 2).

In the *Planning Game* Practice, customers and developers discuss product scope and decide release content and priorities for the next release. Release content is generally captured, although informally, on paper or whiteboard, as ‘Stories’ and so, from a *KiTe* perspective, this Practice defines a *KiTe Method*. The *PlanningGame Method* precondition is ‘empty’ i.e. nothing need be assumed about *Product* prior to *Method* application. During application of this *Method*, some new ‘Stories’ are agreed and the required changes discovered during customer test are also implemented as ‘Stories’. I model as a ‘Planning’ *Activity* with *Method* ‘PlanningGame’. *KiTe* models after application of the ‘Planning’ *Activity* are presented in the

Table 8.5: XP Engineer and Context Models

Attribute	Meaning
Satisfaction	Developer satisfaction with their work (1-10).
Confidence	Developer confidence in their work (1-10).
SubjectAreaKnowledge	Developer knowledge about subject area (1-10).
ProductKnowledge	Developer knowledge about product (1-10)
TechnicalSkills	Subjective measure (1-10) of developer skills
EngineerCommunication	Efficacy of communicn inside dev group (1-10).
CustomerCommunication	Efficacy of communicn between devs and customer (1-10).

second row of Figure 8.10. Some new requirements errors are introduced (*Quality Definition* ‘RemainingDefects’ 3). The result of the *Activity* is an increase in the number of ‘Stories’ defined (*Definition Content* ‘Stories’ 10), the return of the numbers of ‘Story’ defects to zero (*Quality Definition* ‘RemainingDefects’ and *Quality Definition* ‘KnownDefects’) and the subsequent increase in *Quality Definition* ‘RemainingDefects’ as a result of some new injected defects. Both business and technical considerations are taken into account during planning [15] and so this task addresses some architectural concerns. However, results of decisions are not captured, but rather increase *Engineer* ‘ProductKnowledge’ (from 1 to 3) and ‘SubjectAreaKnowledge’ (from 5 to 6). *Engineer* ‘Satisfaction’ increases as a result of the high *Context* ‘CustomerCommunication’ and ‘Confidence’ also remains at a high level. The cost associated with the *Activity* is relatively high as all project personnel are involved.

For *Small Releases*, the rule is that a small number of Stories are implemented in full. This Practice does not in itself change the *Product* and so does not represent a *KiTe Method*. Rather, this Practice constrains ‘Planning’ to output only a small number of complete Stories. Both number of defined Stories and number of new defects is smaller than depicted without *Small Releases*. I model as a *Technique* for *Method* ‘PlanningGame’.

The next Practice is *Metaphor*. This, according to Beck, “replaces much of what other people call ‘architecture’”. Again, this is not captured on paper, but rather provides a common and coherent story for both business and technical people. As for *Small Releases*, the *Metaphor* Practice does not cause a direct change in the *Product*, but in this case the Practice affects how developers approach the design task i.e. design decisions will be constrained by the agreed metaphor. I model the Practice in *KiTe* as a *Technique* applied to design and an increase in engineer understanding of the software to be delivered during *Planning* i.e. in *Engineer* ‘ProductKnowledge’. The situation after *Small Releases* and *Metaphor* is depicted in row three in Figure 8.10.

	Product Definition				Source				Integration				Packaged				Engineer													
	# Stories	Remaining Defects	Known Defects	PersonHours	# Stories	Infrastructure (1-10)	UnitTest	Remaining Defects	Known Defects	Complexity (1-10)	PersonHours	# Stories	Infrastructure (1-10)	UnitTest	Remaining Defects	Known Defects	Complexity (1-10)	PersonHours	SubjectAreaKnowledge (1-10)	ProductKnowledge (1-10)	TechnicalSkills (1-10)	Satisfaction (1-10)	Confidence (1-10)							
Start	3	2	2	1	2	3	2	3	2	3	3	2	3	2	3	2	3	1	2	3	2	3	2	3	0	5	1	5	6	6
PlanningGame	10	3	0	5	2	3	2	3	2	3	3	2	3	2	3	2	3	1	2	3	2	3	2	3	0	6	3	5	8	6
ReleasesMetaphor	7	1	0	5	2	3	2	3	2	3	3	2	3	2	3	2	3	1	2	3	2	3	2	3	0	6	5	5	8	6
PP	7	1	0	5	7	3	7	5	2	5	8	2	3	2	3	2	3	1	2	3	2	3	2	3	0	6	5	5	9	8
PPRefactor	7	1	0	5	7	2	7	5	2	3	9	2	3	2	3	2	3	1	2	3	2	3	2	3	0	6	5	5	9	8
PPRefactorCollective	7	1	0	5	6	2	7	5	2	2	9	2	3	2	3	2	3	1	2	3	2	3	2	3	0	6	6	5	9	8
CodeIntegrate	7	1	0	5	6	2	7	5	2	2	9	6	2	7	5	2	2	3	6	2	7	5	2	2	0	6	7	5	10	9
FunctionalTest	7	1	1	5	6	2	7	5	4	2	9	6	2	7	5	4	2	3	6	2	7	5	4	2	2	6	7	5	10	9
Finish	7	1	1	5	6	2	7	5	4	2	9	6	2	7	5	4	2	3	6	2	7	5	4	2	2	8	9	5	10	9

Figure 8.10: XP process iteration

The *Simple Design* Practice involves creating the minimal design that implements the agreed Stories. However, designs are not generally captured separate from code and so this Practice, as for *Metaphor*, sets a constraint on the code that is produced. The constraint is that only the agreed Stories are represented in the code i.e. there are no frameworks or extra functionality to support future requirements. The constraint introduced with this Practice is that the *Content Source* ‘Infrastructure’ attribute tends towards zero throughout the process.

Beck deals with both ‘unit testing’ and ‘functional testing’ under the single *Testing* Practice. The first relates to running test code to find defects at an implementation i.e. *Source* level. The second relates to user-produced tests to find defects at a specification i.e. at *Definition* level. Both of these change the *Product* and can be considered as possible *KiTe Methods*. I deal with ‘unit testing’ here, and with ‘functional testing’ later in this section.

‘Unit testing’ in XP is carried out as an integral part of the *Pair Programming* Practice. This Practice results in change to the *Product* (some Stories are implemented) and so is also a candidate *KiTe Method*. It also defines how developers should carry out this implementation and thus could represent a *KiTe Technique*. As Pair Programming involves all of design, code and unit test, I introduce a ‘PairProgramming’ *Activity* with *Method* ‘DesignCodeAndUnitTest’. This *Method* results in an increased number of coded Stories (*Content Source* ‘Stories’) and unit tests (*Content Source* ‘UnitTests’). As developers implement the Stories, defects are injected (increase in *Quality Source* ‘RemainingDefects’) but many of these are discovered (increase in *Quality Source* ‘KnownDefects’) and resolved (decrease in both *Quality*

Source ‘RemainingDefects’ and *Quality Source* ‘KnownDefects’). Assuming all discovered defects are resolved, the net effect of the *Method* is a small increase in *Quality Source* ‘RemainingDefects’. The actual number will be dependent upon the capabilities of the engineers, but is also influenced by application of the *Metaphor Technique* as this is said to reduce the risk of making inappropriate design decisions. Engineer capability will also affect adherence to specifications and characteristics of the code i.e. *Content Source* ‘Infrastructure’ and *Content Source* ‘Complexity’. As we have engineers with reasonable technical skills and subject area familiarity and the *SimpleDesign Technique* is applied, I show a zero result for additional infrastructure and a small rise in code complexity. Application of the ‘PairProgramming’ *Activity* is depicted in row four of Figure 8.10. As claimed in the literature, ‘pair programming’ results in increased *Engineer* ‘Satisfaction’ and ‘Confidence’.

The *Refactoring Practice* involves re-organising code to remove any unnecessary code complexity. According to Beck, you “don’t refactor on speculation” but rather “when the system requires that you duplicate code”. He also suggests that taking longer to implement a simpler design is preferable to completing more quickly with a less simple design. This Practice does not change the *Product* in its own right, but again modifies ‘DesignCodeAndUnitTest’ i.e. is another *Technique, Refactor*, for the ‘DesignCodeAndUnitTest’ *Method*. *Quality Source* ‘Complexity’ has reduced from 5 to 3 and cost is slightly higher (row five of Figure 8.10).

The Practice of *Collective Ownership* does not directly change *Product* and so is not a *Method* in its own right. Again it specifies a different *Technique*, ‘CollectiveOwnership’, for ‘DesignCodeAndUnitTest’, because the task now becomes one of focussing on some target code while at the same time changing some of the remaining code base. Revised application of the ‘PairProgramming’ *Activity* with ‘Collective Ownership’ is depicted in row six of Figure 8.10. Application of the Practice effectively reduces velocity i.e. fewer Stories are implemented, but results in a decrease in code complexity in the code base (*Quality Source* ‘Complexity’ 2). There are claims that application of the ‘CollectiveOwnership’ *Technique* increases engineer understanding of the whole system i.e. results in increased *Engineer* ‘ProductKnowledge’.

Continuous Integration is the Practice of integrating and running all unit tests every couple of hours. The idea is that identifying the owner of any introduced defects will be relatively easy as only a small number of code changes are involved. This Practice results in the creation of *Integration* and *Packaged* artifacts and so may be viewed as a *KiTe Method*, say ‘BuildAndUnitTest’. This *Method* requires the existence of *Content Source* ‘Stories’ and results in increased *Content Integration/Packaged* ‘Stories’. It also causes increase to *Quality Integration/Packaged* ‘RemainingDefects’ (as a result of unresolved integration problems) and *Quality Source/Integration/Packaged* ‘KnownDefects’ (as a result of running unit tests). The implication from the XP literature is that each ‘BuildAndUnitTest’ occurrence is immediately followed by ‘fix integration defects’ i.e. problems are not allowed to remain in the integrated code but

are resolved, presumably within minutes. If this truly is the case, the task is better captured as *Method* 'BuildAndUnitTestAndFixProblems' which results in no *Quality Integration* 'RemainingDefects' and no increase in *Quality Source/Integration/Packaged* 'KnownDefects'. I illustrate this situation i.e. an 'Integration' *Activity* with *Method* 'BuildAndUnitTestAndFixProblems'. However, in a real XP project, this might not be the case and the build might remain 'broken' for hours. I look at such a possibility in Section 8.2.10. There are several possibilities for representing the 'Integration' *Activity* in *KiTe*. If carried out as stated in the XP literature, the effect is to closely-couple the 'PairProgramming' and 'Integration' *Activities* i.e. every time a code change is submitted, it is immediately integrated. I represent this by implementing, for each XP iteration, a number of 'Code and integrate' cycles, each comprising *Activities* 'PairProgramming' and 'Integration'. The regular confirmation of code interface correctness causes an increase in *Engineer* 'Confidence' and 'Satisfaction'. The results for the example study after all pair programming-integrate cycles for the iteration are shown in row seven of Figure 8.10. *Source* 'Stories', 'Infrastructure', 'Complexity' and 'RemainingDefects' are reflected in *Integration* and *Packaged* and defect levels remain unchanged. *Engineer* 'Confidence' and 'Satisfaction' increase.

For *FunctionalTesting*, test programs are not part of the product being delivered and so there is a question as to whether or not source and executables for these tests are viewed as attributes of the delivered system. I assume they are. The XP project has no control over the quality and coverage of these tests as these depend upon customer capability, commitment, etc. Functional testing results in an increase in the number of known defects and so is a candidate for a *KiTe Method*. I model as a 'CustomerTest' *Activity* with *Method* 'FunctionalTesting'. The precondition for the 'FunctionalTesting' *Method* is the existence of *Content Packaged* 'Stories'. The results of *Method* application include an understanding by the customer that some of his specifications (Stories) require enhancements or corrections and a discovery of developer-injected defects. The former results in an increase in *Quality Definitions* 'KnownDefects' and the latter in an increase in *Quality Source/Integrations/Packaged* 'KnownDefects'. Both are dealt with in the XP system by a next iteration of Stories. The results of the 'CustomerTest' *Activity* are shown in row eight of Figure 8.10.

The Practice *40 Hour Week* does not cause change to the *Product* or instruct on how any task should be carried out and so is neither a *Method* nor a *Technique*. It is claimed that this Practice results in *Engineer* 'Satisfaction' remaining high. I model as an attribute of *Context* and realise as a lack of decrease in 'Satisfaction', as is generally attributed to working long hours.

On-SiteCustomer results in *Engineer* 'SubjectAreaKnowledge' and 'ProductKnowledge' increasing regularly throughout each iteration. I model as an increase at the end of each iteration.

Coding Standards must be adopted voluntarily by the whole team and are believed necessary for common code ownership. The aim is consistent code quality that facilitates code sharing. As developers are constrained to comply with standards during the ‘PairProgramming’ Activity, I model as a *Technique* to be applied to the ‘DesignCodeAndUnitTest’ *Method*. Application of this *Technique* results in higher code quality, which I model as reduction in *Quality Source/Integration/Packaged* ‘Complexity’.

The final *KiTē Models* at the end of the iteration are shown in the bottom row of Figure 8.10.

For the XP example, attributes for *PS* relate to all of *Content*, *Quality* and *Cost*, attributes for *ES* to engineer skills, knowledge and frame of mind and attributes for *CS* to efficacy of communications.

For *Methods* we have $M = \{‘PlanningGame’, ‘DesignCodeAndUnitTest’, ‘BuildAndUnitTestAndFixProblems’, ‘FunctionalTesting’\}$.

Each *Method* is implemented by a *Technique* that is some amalgamation as shown in Table 8.3. For example, *Method* ‘PlanningGame’ is implemented by the *Technique* that is an amalgamation of the Practices *SmallReleases* and *Metaphor*.

No *GoalsBenchmark* is defined ($GB = \emptyset$).

Discussion

In the *KiTē* representation, *ES* and *CS* capture the belief that the stated attributes (which are typically reported for XP projects) are relevant to project success and other possible attributes are not. This represents an assumption. The implicit statement is that characteristics that have been suggested elsewhere as being relevant, such as ‘privateness’ and ‘dominance’ [3], do not affect success in an XP project. I also note that no *GoalsBenchmark*, *GB*, is defined. *GB* represents the desired ‘finish’ states for the *RealisedProcess* state machine and the implication is that there is an assumption in an XP project that some other means of establishing termination is available.

One point of interest resulting from the above capture in *KiTē* is the variation in meaning of the various Practices. Table 8.6 captures how each Practice relates to a *KiTē* concept. Practices that directly result in a change to the *Product* are marked *Method*, those describing how a task is carried out are marked *Technique*, those affecting how well developers are able to perform tasks are marked *ContextModel* and those affecting process timing and structure are marked *Process*.

Most Practices perform a number of roles. In some cases, Practices that are *Methods* or *Techniques* also have an effect on developer efficacy. For example, ‘Metaphor’ both constrains design and helps increase developer understanding of the product. In other cases, the Prac-

Table 8.6: XP Practices in KiTe

Practice	Method	Technique	ContextModel	Process
Planning Game	Yes	Yes	Yes	No
Small Releases	No	Yes	No	Yes
Metaphor	No	Yes	Yes	No
Simple Design	No	Yes	No	No
Testing (Unit)	Yes	No	Yes	No
Testing (Functional)	Yes	No	No	No
Refactoring	No	Yes	Yes	No
Pair Programming	Yes	Yes	Yes	No
Collective Ownership	No	Yes	Yes	No
Continuous Integration	Yes	Yes	Yes	Yes
40-Hour Week	No	No	Yes	Yes
On-Site Customer	No	No	Yes	No
Coding Standards	No	Yes	No	No

tice describes the *Method* or *Technique* only, for example, ‘Coding Standards’. Although not mentioned specifically as a Practice, ‘TestDrivenDesign’ is another idea expected to be implemented according to Beck. This concept constrains the design task to one of ‘design by unit test creation’. The *KiTe* representation for this is as a *Technique* for the ‘DesignCodeAndUnitTest’ *Method*. The claim is that this *Technique* results in cleaner code design i.e. a reduction in *Quality Source* ‘Complexity’. In the above table, this maps to a ‘Yes’ in the ‘ContextModel’ column for ‘UnitTest’.

From another perspective, I note that the Practices of ‘40-Hour Week’ and ‘Continuous Integration’ are not represented in Table 8.3. These are the only Practices, along with ‘SmallReleases’, represented in the ‘Process’ column of Table 8.6. The Practices are ‘meta-Practices’ in that, rather than add to a description of individual *Activities*, they describe the *RealisedProcess* itself. ‘Small Releases’ states that a single process iteration should be short and the *Realised-Process* comprise many iterations. ‘Continuous Integration’ states a similar fact about build cycles. ‘40-Hour Week’ places a limit on output and cost for each developer and so constrains higher-level management decisions relating to staffing and manpower. This observation raises an interesting research question around introducing these aspects into other, more traditional, environments as their effects would presumably be independent of the specific *Methods* and *Techniques* in place.

An iteration of an XP process comprises a single ‘Planning’ *Activity*, many iterations of ‘PairProgramming’ and ‘Integration’ *Activities* and a single ‘CustomerTest’ *Activity*. ‘Planning’ uses the ‘PlanningGame’ Practice as its *Method* and is constrained by the ‘SmallReleases’

and ‘Metaphor’ Practices as *Technique*. ‘PairProgramming’ has a ‘DesignCodeAndUnitTest’ *Method* and its implementation is constrained by seven of the Practices plus ‘Test Driven Design’. ‘Integration’ uses a ‘BuildAndUnitTestAndFixProblems’ *Method* and this is unconstrained by any specific *Technique*. ‘CustomerTest’ uses the ‘FunctionalTesting’ Practice as *Method* and this is also unconstrained.

The strength of the ‘PairProgramming’ *Activity* in an XP environment would seem to be based on the number of restrictions placed on the implementation of its *Method*. Some interesting research problems might involve identifying a subset of restrictions that might be effective in other environments.

8.2.4 Study 4: Collaborative programming field study

I now represent a field experiment carried out by Nosek to find out what are the effects of collaborative programming i.e. where programmers work in pairs on a common code base [120]. The study concerns a small element (coding) of a traditional waterfall process and involves a small number of programmers. For this representation I use the data supplied by Nosek. The study addresses goals 1.2.1.1. (traditional process), 1.2.2.4. (quantitative study), 1.2.3.3. (small-grained), 1.2.5.1. (industry project), 1.2.6.4. (small project) 1.2.7.1. (CMM level 1), 1.2.8.2. (co-located), 1.2.9.2. (upgrade project) and 1.2.10.1. (standard quality goals).

The aim of the experiment was to examine how developers working in pairs affected outcomes. The experiment used experienced programmers working on an important, challenging program, in their own environment and with their own equipment. The task was to create script files to perform three requirements and a time limit of forty-five minutes was imposed.

Four predictions were made. These were that programmers working in pairs will produce more readable and functional solutions than those working alone, groups will take less time on average, programmers working in pairs will express higher confidence and enjoyment in their work and experienced programmers will perform better. The measured product-related objectives for this study were ‘Functionality’ (up to two points per requirement achieved), ‘Readability’ (the degree to which the problem solving strategy could be determined, measured as 0=unreadable; 2=totally readable) and ‘Time’ (elapsed time to completion in minutes). The measured engineer-related objectives were ‘Confidence’ and ‘Enjoyment’ (no scale given).

Results for the control group (individual programmers) and the experimental group (pairs) are shown in Table 8.7. Means are given, with standard deviations in brackets.

Results supported predictions 1, 2 and 4 with probability of less than 1 in 20 (5 percent confidence) that results were due to chance. The prediction relating to ‘ElapsedTime’ was not statistically supported as there was more than a 1 in 20 chance the result was due to chance.

Table 8.7: Collaboration results (Nosek)

Output variable	Control (individuals)	Experimental (pairs)
Functionality	4.2 (1.788)	5.6 (0.547)
Readability	1.4 (0.894)	2.0 (0.000)
ElapsedMinutes	42.6 (3.361)	30.20 (1.923)
Confidence	3.8 (2.049)	6.50 (0.500)
Enjoyment	4.00 (1.870)	6.60 (0.418)

KiTe representation

In this reproduction, I address the first three predictions only i.e. programmers working in pairs will produce a more readable and functional solution in less time. The reason is that the reported study contains no information or data to support the prediction that experienced programmers perform better, and the indication is that the mix of experienced and inexperienced is held constant between the the pairs and individuals.

The first task is to define an appropriate *Product* model. I model *Content* ‘Functionality’, *Quality* ‘Readability’ and *Cost* ‘ElapsedMinutes’. However, it is not clear from the study what are the inputs to *Method* (for example, requirements documents, designs, verbal instruction) and not clear what are the outputs (for example, script sources, integrated and tested scripts, scripts plus design documentation, etc.). This means I do not know which *Partitions* to include in my model for *Product*. In order to illustrate capture, I assume requirements are inputs and outputs are script sources.

Table 8.8: Collaboration Product Model

Perspective	Partition	Attribute	Meaning
Content	Definition	Requirements	# documented requirements
	Source	Functionality	0-6; 2 for each of 3 implemented requirements
Quality	Source	Readability	0-2; 0=unreadable and 2=totally readable
Cost	Source	ElapsedMinutes	Time in minutes

I call the *Method* for this study, ‘CodeRequirements’. We have $M = \{\text{CodeRequirements}\}$. The product-related precondition and transformation that define this *Method* are shown in Table 8.9.

I now must decide how to represent the ‘Collaboration’ and ‘Individual’ factors that are the main factors-of-interest for the study. An obvious choice is to model as a *Context* i.e. as affecting how well developers are able to carry out coding. My reason for considering this as a

Table 8.9: Collaboration CodeRequirements Method

CodeRequirements	
Precondition	<i>Content Definition</i> ‘Requirements’ == 3)
Transformation	<i>Content Source</i> ‘Functionality’ >= 0 and <= 6; <i>Quality Source</i> ‘Readability’ >= 0 and <= 2; <i>Cost Source</i> ‘ElapsedMinutes’ >= 0 and <= 45.

possible option is that the study of how developers work in teams has tended to be considered as a study of project context. For this choice, *ContextModel* would contain some ‘rules’ about the effects of working in pairs. As a *Context*, these would be applied with no ‘matching’ to engineer characteristics. However, I remember that some developers are more suited to working collaboratively than others, and so would like to find a representation that takes this into account. I can achieve this by modelling ‘Collaboration’ and ‘Individual’ as *Techniques*. *ContextModel* now ‘matches’ the *Technique* requirement (for example, ‘work collaboratively’) with *Engineer* characteristics (for example, ‘introverted’) to achieve a result that takes individual engineers into account.

The above means that I should represent factors as *Context* factors only if I believe that individual engineer characteristics are irrelevant. For example, I might model ‘Company about to be bought’ as a context that causes engineers to become less motivated because I believe that I may treat engineers in a general way. In the case under study, there is a strong possibility that engineers differ in their preference for working with people, and so I represent ‘Collaboration’ and ‘Individual’ as two *Techniques* to be compared.

For *Engineer*, I include ‘Confidence’ and ‘Enjoyment’, each using an ordinal scale of 1-10, and ‘Experience’ measured as ‘NumberOfYears’. Nosek held constant familiarity with environment and equipment and unfamiliarity with kind of problem and I add these attributes to the model for *Engineer*. There is no mention of any other context-related factors. *Engineer* and *Context* models are presented in Table 8.10. Nosek assumes results are due to the application of the *Techniques* under study only and so *ContextModel* does not contribute to the result.

Experimental results from a *KiTe* perspective are depicted in Figure 8.11. The first row captures the set of *ProductMeasurement* for start state $ps0 \in PS$ and the *CapabilitySpec* that characterises $es \in ES$ prior to application of ‘CodeRequirements’. The second row illustrates end states for *Technique* ‘Individual’ and the third for *Technique* ‘Pairs’. Values for ‘Experience’, ‘EnvironmentFamiliarity’ and ‘ProblemUnfamiliarity’ are empty because no indication was given as to what the values should be, possibly because these were held constant. Mean values only are captured and this is discussed in the next Section.

Table 8.10: Collaboration Engineer and Context Model

	Attribute	Measure
Engineer	Confidence	1-10
	Enjoyment	1-10
	Experience	Number of years
	EnvironmentFamiliarity	
	ProblemUnfamiliarity	
Context		

	Product Definition	Source			Engineer				
	Requirements	Functionality (0-6)	Readability (0-6)	ElapsedMinutes	Confidence (1-10)	Enjoyment (1-10)	Experience (1-10)	EnvironmtFamiliarity (1-10)	ProblmUnfamiliarity (1-10)
Start	3	0.00	0.00	0.00	0.00	0.00			
Individual	3	4.20	1.40	42.60	3.80	4.00			
Collaboration	3	5.60	2.00	30.20	6.50	6.60			

Figure 8.11: Collaboration results in KiTe

I now examine results to establish relative values for performance for the two *KiTe Activities* associated with the *Techniques* ‘Individual’ and ‘Collaboration’. I notice that the expected output from the ‘CodeRequirements’ *Method* with baseline *Technique* was a score of ‘6’ for ‘Requirements’ within 45 ‘ElapsedMinutes’ with a ‘Quality’ score of ‘2’. Working with mean values only, I calculate that, for the ‘Individual’ *Technique*, the relative effectiveness is $\{('Functionality', 0.7 (4.2/6)), ('Readability', 0.7 (1.4/2)), ('ElapsedMinutes', 1.06 (45/42.6))\}$.

The same calculations for ‘Collaboration’ yields $\{('Functionality', 0.93 (5.6/6)), ('Readability', 1 (2/2)), ('ElapsedMinutes', 1.49 (45/30.2))\}$.

Because the contribution from *ContextModel* is assumed to be of unit size, the implication is that the results are due entirely to the two *Techniques*.

Discussion

Once more, in the attempt to capture the experiment in *KiTe*, some lack in clarity was made visible. For *Method*, it is unclear what is the pre-condition (requirements, designs, word-of-mouth, etc.) and what are the deliverables (script sources, integrated and tested scripts, scripts plus design documentation, include tests and test results, etc.). This means we cannot define the model for *Product*. In addition, no mention is made of any other techniques to be implemented. For example, a technique ‘collaborating using structured programming’ may yield different results from ‘collaborating using an ad-hoc approach’. For *Engineer*, we are informed of a number of attributes believed to be of relevance. The first point to note is that, for some of these, for example, ‘Experience’, we are informed that these were held constant but are not provided with any indication of values. This means that we can not compare this study with any other, apparently similar, study in case different levels of experience apply. The observation is that values must be reported even if held constant. The second point to note is that there is an assumption is that no other attributes affect outcomes. There is no mention of any context-related factors and this equates to an assumption that such factors do not affect outcomes or affect each study in the same way.

I note from above two ideas that must be further investigated. The first is that I have captured mean values only and the statistical aspect of the experiment is lacking. I discuss this further in Sections 10.5 and 11.3. The second is the observation that any factor that changes *Engineer* that might be dependent upon individual engineer characteristics is better modelled in *KiTe* as a *Technique*. This represents a subtlety of *KiTe* that I believe adds to its power.

8.2.5 Study 5: Event-driven simulation model

In this Section, I study a simulation model created by Melis et. al. [109] for the investigation of the XP practices of Pair Programming (PP) and Test Driven Development (TDD). The study thus addresses Goals 1.2.1.2. (agile process), 1.2.2.1. (simulation model), 1.2.3.3. (small-grained), 1.2.4. (variations to base process) and 1.2.6.4. (small project).

The simulation model is essentially ‘event-driven’. It has a number of ‘entities’, for example, ‘user stories’ (US) and ‘integrated code’, and some ‘activities’, for example, ‘release planning’ and ‘development session’, that modify the ‘entities’. ‘Activities’ are carried out by ‘actors’, for example, ‘team’, ‘developer’, ‘customer’. ‘Actors’ have attributes that change with time. ‘Entity’ modification occurs at the end of an ‘activity’ (a time step), at which time continuous variables, for example, ‘developer skill’, are calculated using integration rates. To handle uncertainty, for example, effort estimates for ‘user stories’, a stochastic approach is used, and statistical distributions and Monte Carlo simulation applied. The simulation shows that the use

of pair programming increases development cost (working days), improves quality (smaller defect density) and design (fewer lines of code per US). The use of test driven design increases project duration and decreases defect density.

The granularity of the simulation is a development session, typically a couple of hours. Model equations are taken from existing models, empirical data and author assumptions, where necessary. Inputs are ‘number of initial US’, ‘number of developers’, ‘mean and standard deviation of initial US estimation’, ‘initial team velocity’, ‘number of iterations per release’ and ‘typical iteration duration’. Outputs are ‘number of final US’, ‘defect density’, ‘number of classes and methods’ and ‘delivered source instructions (DSI)’.

The model allows the user to define the adoption level of PP and TDD practices. In Table 8.11, results are given for the four combinations of zero and full adoption of each practice. Standard deviations are reported in parentheses.

Table 8.11: PP and TDD adoption results (Melis et. al.)

Output variable	PP=0% TDD=0%	PP=100% TDD=0%	PP=0% TDD=100%	PP=100% TDD=100%
Working days	45.0 (23.2)	51.1 (19.1)	51.1 (23.6)	60.3 (22.8)
Released US	28.8 (7.9)	28.8 (7.6)	28.7 (7.6)	28.9 (7.5)
Defects/KDSI	28.0 (5.3)	24.1 (6.0)	23.0 (5.3)	19.7 (4.5)
KDSI	18.0 (8.2)	13.0 (6.1)	21.5 (10.2)	15.6 (6.9)

KiTe representation

In order to create the model for *Product*, I must identify the *Perspectives* for this simulation along with the attributes-of-interest for each *Perspective*. However, the paper presents some difficulties with attribute meaning and I must deal with the first set of assumptions. For example, ‘defect’ is used without any definition of what is a ‘defect’. Basili and Rombach use *defect* as a generic term to mean any one of *error*, *fault* or *failure* [13, 71], but no such definition is given in this paper. A similar comment applies to ‘KDSI’ and ‘Working days’. A more serious problem occurs with the use of ‘KDSI’. In the ‘Results’ section, we are informed that “the use of PP decreases the number of DSI by 27%”. The implication is that ‘DSI’ is a quality-related metric i.e. code is more succinct when applying pair programming. There are two issues. The metric ‘KDSI’ (thousand delivered source instructions) is often used as a measure of *Content* i.e. as indicator of how much work has been done. This is quite a different meaning and any attempt to compare the results of experiments on the grounds of ‘KDSI’ might fail if the meanings are not clear. The second issue is related. Results in Table 8.11 show that the number of delivered

lines of code decreases when pair programming is applied and increases when test driven design is applied. The implication is either that test driven design decreases code quality or increases the code output. The latter would seem more likely, and if this is the case the authors are now implying that increased code represents more content (tests), rather than reduced quality. It is meaningless for an attribute to have two different interpretations, and questions must be asked about what exactly is being measured. For this illustration, I simply note the problem and select the quality-related meaning.

To complete the model for *Product*, I must consider what are the *Methods* for the simulation and how these affect *Product*. There are four simulation scenarios:

- No Pair Programming (PP); no TestDriven Design (TDD).
- All developers carry out PP; no TDD.
- No PP; all developers carry out TDD.
- All developers carry out PP; all developers carry out TDD.

Table 8.12: PP and TDD Product Model

Perspective	Partition	Attribute	Meaning	
Content	Definition	US	# Stories captured	
	Source	US	# Stories implemented	
	Integration	US	# Stories integrated	
	Packaged	US	# Stories delivered	
Quality	Source	KDSI	thousands delivered source instructions	
		DefectsPerKDSI	# defects per thousand DSI	
	Integration	KDSI	thousand delivered source instructions	
		DefectsPerKDSI	# defects per thousand DSI	
	Packaged	KDSI	thousand delivered source instructions	
		DefectsPerKDSI	# defects per thousand DSI	
	Cost	Source	WorkingDays	Total person days

When I attempt to capture *Methods*, I find the problem relating to test cases discussed above creates a problem. If the scenarios with ‘no TDD’ result in no test cases, we have two different *Methods* because the outputs are different. This means that, from a *KiTe* perspective, we may not directly compare the ‘no TDD’ simulations with the ‘TDD’ ones. Comparisons would be unhelpful because more output is being produced in the ‘TDD’ cases. However, as the simulation directly compared the four scenarios, and in order that I might illustrate capture, I simply note the lack of clarity and assume outputs are the same in all cases. I implement a single ‘DesignAndCodeAll’ *Method* and apply four different *Techniques*, one for each of the

scenarios above. I also assume, as for the earlier XP studies, that *Integration* and *Packaged* outputs are delivered ‘for free’ (see Section 8.2.3). The model for *Product* is shown in Table 8.12.

Table 8.13: PP and TDD DesignAndCodeAll Method

DesignAndCodeAll	
Precondition	<i>Content Definition</i> ‘US’ > 0
Transformation	<i>Content Source/Intergation/Packaged</i> ‘US’ increases; <i>Quality Source/Integration/Packaged</i> ‘KDSI’ increases; <i>Quality Source/Integration/Packaged</i> ‘Defects/KDSI’ increases; <i>Cost Source/Integration/-Packaged</i> ‘WorkingDays’ increases.

Method is ‘DesignAndCodeAll’ i.e. all *Definition* ‘US’ are to be implemented. We have $M = \{\text{DesignAndCodeAll}\}$. The product-related precondition and transformations that define this *Method* are shown in Table 8.13.

The *Techniques* that are being compared in this simulation are:

- no PP; no TDD.
- 100% PP; no TDD.
- No PP; 100% TDD.
- 100% PP; 100% TDD.

I use the first as the baseline case i.e. the expected result against which to compare other results is 28.8 ‘US’ are delivered in 18.0 ‘KDSI’ with 28.0 ‘Defects/KDSI’ and taking 45.0 ‘WorkingDays’. The results as represented in a *KiTe Product* model are reproduced in Figure 8.12. *Source* values for *Content* and *Quality* are propagated to *Integration* and *Packaged* but the value for ‘WorkingDays’ is ‘0’ for these *Partitions* (see discussion in Section 8.2.3). Each row in the table represents the *Product* end state for each of the four *Techniques*. The relative performance values for each of the *Techniques* is presented in Figure 8.13.

Individual developer characteristics and contexts are not mentioned and I assume models as for the earlier XP studies (see Section 8.2.3).

Capture in *KiTe* reflects the result that, assuming all differences are due to the *Techniques* implemented, pair programming is more effective in that it produces less code (factor of 1.38) and causes fewer defects (factor of 1.16), but is less ‘time effective’ (factor of .88) in that it takes longer.

	Product Source				Integration				Packaged			
	US	KDSI	Defects/KDSI	WorkingDays	US	KDSI	Defects/KDSI	WorkingDays	US	KDSI	Defects/KDSI	WorkingDays
No PP; No TDD	28.80	18.00	28.00	45.00	28.80	18.00	28.00	0.00	28.80	18.00	28.00	0.00
100% PP; No TDD	28.80	13.00	24.10	51.10	28.80	13.00	24.10	0.00	28.80	13.00	24.10	0.00
No PP; 100% TDD	28.70	21.50	23.00	51.10	28.70	21.50	23.00	0.00	28.70	21.50	23.00	0.00
100% PP; 100% TDD	28.90	15.60	19.70	60.30	28.90	15.60	19.70	0.00	28.90	15.60	19.70	0.00

Figure 8.12: PP and TDD simulation results in KiTe

	Product Source				Integration				Packaged			
	US	KDSI	Defects/KDSI	WorkingDays	US	KDSI	Defects/KDSI	WorkingDays	US	KDSI	Defects/KDSI	WorkingDays
No PP; No TDD	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
100% PP; No TDD	1.00	1.38	1.16	0.88	1.00	1.38	1.16	0.88	1.00	1.38	1.16	0.88
No PP; 100% TDD	1.00	0.84	1.22	0.88	1.00	0.84	1.22	0.88	1.00	0.84	1.22	0.88
100% PP; 100% TDD	1.00	1.15	1.42	0.75	1.00	1.15	1.42	0.75	1.00	1.15	1.42	0.75

Figure 8.13: PP and TDD Technique relative performance

Discussion

Capture in *KiTe* requires that the model for *Product* for the study is clearly defined. This means that appropriate *ProductPerspectives* are selected and items of interest within these perspectives are defined with measures clearly stated. For this simulation experiment, it is not clear how the metric ‘defects per thousand lines of code’ is measured, what a ‘defect’ is or what is meant by a ‘working day’. These uncertainties are effectively assumptions embedded in the model i.e. the model is build on items and equations that use a specific meaning of ‘defect’ and a specific way of measuring ‘defects per thousand lines of code’. Another consideration with *Product* model definition is that of the meaning of ‘KDSI’ which seems to be used both as a quality indicator and a measure of size.

I next observe a number of uncertainties in the definition of the selected *Method* and *Techniques*. For example, it is not clear if developers create unit tests in all cases.

Another source of assumptions relate to engineer and context. In Figure 8.13, I implied all

of the variation in results is due to to the relative effect of *Technique*. It is possible that some of this variation would have been due to differences in contexts, for example, differences of developer skill and motivation. In fact, if we examine the ‘improvement’ in duration when pair programming is not used, we find two results. When TDD is actioned, the result with no PP is 15 percent better (51.1/60.3). When TDD is not actioned, the result is only 12 percent better (45/51.1). As these calculations are based on means only, it is possible that such a result in a real-life study would indicate no inconsistency. It would also, however, be possible that the result is an indication that something other than pair programming is affecting results. The *KiTe* approach would be to separate the *Technique*-related data (‘PP is on average 40 percent more time effective’) from the context-related data (‘the context model assigns average pair velocity and effectiveness of the most skilled developer’). This calculation model is hidden in the results as presented, but would become transparent if captured as a *KiTe ContextModel*.

As for other studies, the attempt to represent this simulation in *KiTe* reveals a lack of clarity in the description of the simulation. Product attributes are undefined and treated inconsistently, making it difficult to define the model for *Product*. Lack of clarity about outputs from the four scenarios means that it is difficult to capture *Method* with any certainty and so it is not clear that direct comparison of results is appropriate.

8.2.6 Study 6: Pair programming classroom study

This study concerns a classroom experiment to investigate whether pair programming results in code being developed faster and with better quality [169]. The study addresses goals 1.2.1.2. (agile process), 1.2.2.4. (quantitative study), 1.2.5.2. (student project), 1.2.6.5. (tiny project) and 1.2.10.1 (standard goals).

The experiment was carried out in 1999 by Williams et. al.. Forty-one senior software engineering students were divided into an experimental group and a control group (individual programmers). The task was to complete four assignments using a pair-programming approach. The two groups comprised the same mix of high, average and low performers. “All students attended the same classes, received the same instruction and participated in class discussions on the pros and cons of pair programming.” Groups completed four assignments over a period of six weeks.

The experiment compared the cycle time, productivity and quality between the two groups. Results showed that the pairs always passed more of the post-development test cases and results were more consistent. Pairs completed assignments 40-50 percent faster i.e. with only a small drop in productivity.

KiTe representation

I now attempt to capture the study in *KiTe* and show that this effort also exposes ambiguities and assumptions in the target study.

Product

Content: The deliverables from the process were four assignments, but we have no information about what kind of assignment (for example, stand-alone software application executables, software component source code, analysis report, etc.). This means we do not know what the pair programming was applied to.

Quality: The experiment reports ‘percentage of test cases passed’ and this is an operationalisation for a *Quality* measure. However, there is no discussion about the focus and breadth of test cases, for example, kinds of defects tested for, test coverage, etc. and so we do not know what the reported metric means. This means we will not be able to compare this experiment with other, apparently similar, studies.

Cost: The experiment reports ‘completion times as a percentage’.

Method

Preconditions: The first aim for defining *Method* involves clarifying what is the expected state of the *Product* prior to *Method* invocation. A ‘pair programming’ task might be based on any of formal specifications, informal user stories, design documents, discussions with the customer, etc. In *KiTe*, each of these potentially represents a different *Method*. The form of the specifications for this experiment is not mentioned.

Transformation: I next consider how the *Method* changes the *Product*. As stated above, we are not told if the expected deliverables are software source or executables or written reports. Even if we assume standalone software application executables, there is still some uncertainty as to what the *Method* involves. We understand some integration and ‘packaging’ for delivery is indicated but we are not clear about whether the students simply build locally, and this creates the software to deliver, or whether some integration into a larger system is required, indicating some more complex build process.

Technique: Last of all, we must know exactly what techniques are being implemented. The paper describes the XP pair programming ‘rules’ but does not state that the experimental groups were instructed to follow, or checked for compliance with, these rules. Did pairs actively engage in the same code at the same time, swap drivers, etc.? If they did not, we are testing something other than ‘pair programming’.

ContextModel

Engineers: We are informed that all students received the same instruction and that each group contained a similar mix of capabilities. The *ContextModel* for the experiment thus involves ‘capabilities’. ‘Instruction’, however, is neither a *Context* or an *Engineer* attribute. Presumably ‘level of instruction’ is believed to relate to ‘level of expertise’ in some way. The *Context* model includes no other *Engineer*-related attributes and so other attributes, for example, motivation are believed to be of no consequence. In fact, for this experiment, all of the experimental group and seven of the control group had a preference for pair programming and it is very likely that this would skew results. In addition, there may have been some bias due to the researchers having an XP focus and being in positions of power over the students. Any of these would probably affect motivation levels and cause the results to be scaled towards supporting pair programming.

Contexts: The study does not consider any other contextual factors. The complete *ContextModel* thus includes only *Engineer* capabilities and expertise.

Discussion

This study presents results in an informal way and, because of this, I use the study mainly as a means of exposing assumptions. In Section 8.3.1, I make decisions for some of the assumptions in order to ‘fix’ models and include the study in a comparison between various collaborative approaches.

8.2.7 Study 7: State-based simulation model

I next consider a simulation model created to provide a quantitative analysis of the results of improving inspections in a company that implements a process based on a waterfall model. The study addresses goals 1.2.1.1. (traditional process), 1.2.2.1.2 (state based simulation model), 1.2.3.3. (small-grained process), 1.2.4. (variations to base process), 1.2.5.1. (industry project), 1.2.6.3. (medium size project), 1.2.7.1. (CMM level 1), 1.2.8.2. (co-located project), 1.2.9.2. (upgrade project) and 1.2.10.1) standard goals.

This proof-of-concept feasibility study is based on work by Raffo, Vandeville and Martin [137] from the School of Business Administration, Portland State University, Oregon, US. The cited paper describes a simulation model developed for Northrop Grumman under sponsorship of the Software Engineering Research Center (SERC). The model is one of a number of stochastic models built to provide a quantitative understanding of the development process and support the quantitative analysis of proposed process changes prior to implementation. The example

presented in the paper examines quality outcomes when the review and inspection processes are improved.

Some key aspects of the feasibility study are:

- The development process at Northrop Grumman's Melbourne site comprises a number of hierarchical process components and the portion modelled consists of four life cycle phases - Preliminary Design, Detailed Design, Code And Unit Test, Computer Program Engineering Test (*CPET*).
- Each lifecycle phase decomposes into a number of main tasks and sub-tasks. Each sub-task is a distinct development step with associated process performance data.
- There are five major product verification points. Three are product reviews (task architecture inspection, unit architecture inspection, code walkthrough) and two are testing activities (unit test and process test).
- The data used to populate the model were taken from the various project teams on a large upgrade project (200,000 lines of code, existing system 2 million lines of code).
- The target project uses a number of integrated development teams. The model simulates a single development team executing the standard development process.
- The total lines of code for the project is 10,000 and the total number of defects typically produced for this amount of development is 500 defects.
- It is assumed that 20 percent of total errors are injected during task architecture, 30 percent during unit architecture and 50 percent during coding.
- It is assumed that the cost of fixing defects is .5 hours per error for errors found during task architecture inspection, 1 hour per error for errors found during unit architecture inspection, and 2, 4 and 6 hours per error for those found during code walkthrough, unit test and process test, respectively.
- The error detection capability is .3 of current product errors for the reviews and .75 for the test activities. The purpose of the simulation is to examine the result on residual errors when the review detection capability is raised to .7.
- The model provides the mean and variance on performance results i.e. the results are stochastic.

The simulation model used is a state based model based on work by Kellner and others at the Software Engineering Institute (*SEI*) in the mid-80s. The developed model uses cost,

Table 8.14: State-based simulation Product Model (Raffo et. al.)

Perspective	Partition	Attribute	Meaning
Content	Definition	Requirements	% requirements defined
	Architecture	ContentComplete	% requirements architected
	Design	ContentComplete	% requirements designed
	Source	ContentComplete	% requirements implemented
	Integration	ContentComplete	% requirements integrated
Quality	Architecture	DefectsRemaining	# known + # undiscovered arch defects
		DefectsDetected	# known architected defects
	Design	DefectsRemaining	# known + # undiscovered desn defects
		DefectsDetected	# known designed defects
	Source	DefectsRemaining	# known + # undiscovered impl defects
		DefectsDetected	# known implemented defects
	Integration	DefectsRemaining	# known + # undiscovered integr defects
		DefectsDetected	# known integrated defects
Cost	Architecture	PersonHours	Time to architect
	Design	PersonHours	Time to design
	Source	PersonHours	Time to implement
	Integration	PersonHours	Time to integrate

schedule and quality data from past projects and applies statistical methods to analyse outputs. The research goal of the feasibility study is to determine the most suitable statistical techniques to deal with sparse and correlated data.

KiTe representation

Product The study focusses on software quality, and the quality focus is ‘number of remaining errors’. There is also an interest in ‘number of discovered errors’. Definitions for the various kinds of defects that occur in software products are provided by several authors (see [13, 71]), and Basili and Rombach define *error* as “...defects in the human thought process made while trying to understand given information, to solve problems, or to use methods and tools” [13]. As it is not clear that this is the meaning intended by Raffo et. al., I use the more general ‘defects’. The cost attribute described is effort measured in person hours. The target simulation appears to assume 100 percent functionality is implemented throughout.

I define *Content*, *Quality* and *Cost Perspectives*. As the simulation assumes the existence of functional requirements and implements a process from ‘task architecture’ through to ‘internal integration and testing’, *Product* will include the *Partitions Definition*, *Architecture*, *Design*, *Source* and *Integration*. The model for *Product* is presented in Table 8.14.

Raffo et. al provide effort totals for each phase and rates for rework, but data is not provided

for some tasks, for example, architecture and inspections. This means I do not have enough information to represent project effort and so I focus below on *Quality* only.

ContextModel In this study, no mention is made of any engineer characteristics or context factors that might affect results. *ContextModel* thus does not modify *Activity*'s transformation and effects no change to *Engineer ES* or *Context CS*.

Methods From the paper, I identify the following Methods. In doing so, I identify some instances of lack of clarity in *Method* description. These involve the *Methods* relating to defect resolution. As discussed in Section 8.2.1, the failure to state what is the policy for defect resolution is a common problem with traditional software process descriptions. There are a number of possibilities for such a *Method*, for example, defects in *Source* artifacts may be resolved in code only, in designs and code, etc. In this paper, I capture the situation of local resolution only e.g. defects found in designs are resolved in designs only. For the last step in the process, resolution of integration defects, I assume defects in earlier phases are resolved. The reason is one of pragmatism — the required data is not available, and so I choose the simplest option.

DefineRequirements Needed only to give us an initial value for Definitions.

AnalysisAndHighLevelArchitecture Analyse software requirements and architect at high level (task architecture).

InspectArchitectures Task architecture inspection.

ResolveArchitectureDefectsInArchitectures Resolve known defects in Architecture artifacts.

DetailedDesignsFromArchitectures Detail design based on Architecture artifacts (architect at software unit level).

InspectDesigns Unit architecture inspection.

ResolveDesignDefectsInDesigns Resolve known defects in Design artifacts.

CodeFromDesigns Create source code based on Design artifacts.

InspectCode Code walkthrough.

ResolveCodeDefectsInSources Resolve known defects in Source artifacts.

UnitTestCode Unit test code.

ResolveCodeDefectsInSources Resolve known defects in Source artifacts.

IntegrateAndTest Internal integration and test.

ResolveIntegrationDefects Resolve known defects in Integrations in Architecture, Design, Source and Integration artifacts.

	Arch		Design		Source		Integrn	
	<i>DefectsRemaining</i>	<i>DefectsDetected</i>	<i>DefectsRemaining</i>	<i>DefectsDetected</i>	<i>DefectsRemaining</i>	<i>DefectsDetected</i>	<i>DefectsRemaining</i>	<i>DefectsDetected</i>
Start	0	0	0	0	0	0	0	0
AnalysisAndHighLevelArch	100	0	0	0	0	0	0	0
InspectArchitectures	100	30	0	0	0	0	0	0
ResolveArchitectureDefectsInArchitectures	70	0	0	0	0	0	0	0
DetailedDesignsFromArchitectures	70	0	250	0	0	0	0	0
InspectDesigns	70	0	250	83	0	0	0	0
ResolveDesignDefectsInDesigns	70	0	167	0	0	0	0	0
CodeFromDesigns	70	0	167	0	500	0	0	0
InspectCode	70	0	167	0	500	167	0	0
ResolveCodeDefectsInSources	70	0	167	0	333	0	0	0
UnitTestCode	70	0	167	0	333	250	0	0
ResolveCodeDefectsInSources	70	0	167	0	83	0	0	0
IntegrateAndTest	70	0	167	0	83	62	83	62
ResolveIntegrationDefects	70	0	167	0	21	0	21	0

Figure 8.14: State-based simulation baseline

Because the number of *Methods* is large, I do not detail preconditions and effects for each *Method*. Assuming a traditional waterfall, preconditions for ‘AnalysisAndHighLevelArchitecture’ would be availability of completed requirements and for ‘InspectCode’ would be availability of completed code. *Method* effects for *Methods* are, for example, completed code and defect injection for ‘CodeFromDesigns’. Assumptions have been made above for *Methods* involving defect resolution.

I note that another lack of clarity is uncovered when attempting to define *Methods* and *Method* ordering. From the results presented in Fig. 4 in the target paper, I understand that ‘code walkthrough’ occurs before ‘unit test’. I implement ‘InspectCode’ for the walkthrough, followed by ‘ResolveCodeDefects ...’. However, it would seem reasonable to assume that ‘ResolveCodeDefects ...’ is carried out after both ‘code walkthrough’ and ‘unit test’. Perhaps, however, some other strategy is followed, for example, many instances of ‘walkthrough’ and ‘unit test’ intermingled as required. In any case, what actually takes place is not clear from the paper and this is discussed further below.

The study involves implementing a single pass of the *Methods*. No specific techniques are mentioned in the study. However, the aim of the simulation is to examine the impacts on process

performance if review effectiveness is increased. This is synonymous with replacing one review *Technique* with a ‘better’ one.

The baseline case for the simulation may be inferred from the assumptions about the ‘expected’ error injection rates for ‘design’ and ‘code’ *Methods* and about review effectiveness rates for the ‘review’ and ‘test’ *Methods*. As the study assumes all outcomes are a result of the implemented *Methods* and *Techniques*, these rates are a statement of the expected performance of these *Methods* and *Techniques*. From Fig. 3 in the paper, I note that assumed error injection rates are 20 percent for architecture, 30 percent for design and 50 percent for coding, and the total errors injected is 500. The baseline values are thus 100 errors for ‘AnalysisAndHighLevelArchitecture’ (1 per percent requirements = $0.2 * 500 = 100$ errors), 150 errors for ‘DetailDesignsFromArchitectures’ and 250 for ‘CodeFromDesigns’. As-is detection capabilities are assumed to be .3 for ‘InspectArchitectures’, ‘InspectDesigns’ and ‘InspectCode’ and .75 for ‘UnitTestCode’ and ‘IntegrateAndTest’. It is assumed that errors injected during testing are negligible. I illustrate the simulated process for the baseline case in Figure 8.14.

Defects are injected during architecture, design, etc., discovered by reviews and tests and then resolved. As discussed above, I illustrate a situation where defects are resolved locally only, with the exception of the final step, ‘ResolveIntegrationDefects’, where defects are resolved in earlier artifacts also.

Effectiveness I now use the data in Figs 4 and 5 in the paper to calculate effectiveness relative to the baseline for each *Method* as implemented in the as-is and to-be simulations. Effectiveness values for the as-is process are calculated in Table 8.15 and for the to-be process in Table 8.16.

The calculated values describe the effectiveness of the *Activities* based on the *Methods* presented above. Summaries of *Activity* effectiveness values for the as-is and to-be simulations are presented below in Table 8.17.

Discussion

Our aim was to capture the process by selecting *Methods* and defining relative *Method* (and *Technique*) performance values and *ContextModel* adjustments representing project contexts. Some questions and observations arise:

- It’s not clear why the rework rates for defects found during code walkthrough (2 hours per error) and unit test (4 hours per error) are so different. Code walkthrough occurs after coding is complete and then is immediately followed by unit testing. So why the difference in rework rate? Presumably there is another process step between these? Or some test-related procedures that are ‘expensive’?

Table 8.15: State-based As-Is calculations

Method	Defects	Effectiveness
Analyse....	Errors injected 99 (27.3+71.7 - detected + escaped). 'Quality DefectsRemaining' 99.	100/99=1.01
InspectArchitectures	Errors discovered 27.3. 'Quality DefectsDiscovered' 27.3.	.276 (27.3/99)
ResolveArchDefects...	'Quality DefectsRemaining' 71.7.	1
DetailDesigns...	On completion, 221.2 defects remain (61.4+159.8) i.e. 'Quality DefectsRemaining' 221.2. We infer that 221.2-71.7 = 149.5 defects were injected	150/149.5 = 1.003
InspectDesigns	Of the 221.2 defects remaining, 61.4 are detected. We now have Remaining=221.4; Detected=61.4.	.278
ResolveDesnDefects...	61.4 defects resolved in Design documents leaving 221.2-61.4=159.8 remaining. We don't have enough information to know how many of those fixed origi- nated in, and were fixed in, Architecture documents. We will assume the Method fixes Design defects only.	1
Code...	On completion, 409.3 defects remain (114.4+294.9) i.e. 'Quality DefectsRemaining' 409.3. We infer that 409.3-159.8=249.5 were injected	1
InspectCode	Of the 409.3 defects remaining, 114.4 are detected. We now have Remaining=409.3; Detected=114.4.	.28 (114.4/409.3)
ResolveCodeDefects...	114.4 defects are resolved in code, leaving 409.3- 114.4=294.9 remaining. We don't know how many of these originated in Architectures and Designs and so we will assume the Method fixes code defects only. We also see that from table 4 after test the number is 304.2. 10.3 new errors were injected during fixing. So this Activity resulted in 114.4 resolved errors, 10.3 new ones injected.	104.1/114.4=.91.
UnitTestCode	Of the 304.2 defects remaining after unit test, 191.9 were detected	191.9/304.2=.63
ResolveCodeDefects....	191.9 defects resolved in code, leaving 304.2- 191.9=112.3 remaining. Again, we don't know how many of these originated earlier on. However after process test, 121.4 defects remained (81.6+39.8). So this Method caused 9.1 defects to be injected. 'De- fectsRemaining'= 121.4.	182.8/191.9=.95
IntegrateAndTest	Of the 121.4 defects remaining, 81.6 were detected	.67
ResolveIntegrnDefects	81.6 defects are resolved, leaving a DefectsRemain- ing of 39.8. All fixed.	1

Table 8.16: State-based To-Be calculations

Method	Defects	Effectiveness
Analyse...	Errors injected 99 (64.2+34.8 - detected + escaped). 'Quality DefectsRemaining' 99..	100/99=1.01
InspectArchitectures	Errors discovered 64.2. Quality DefectsDiscovered' 64.2.	.65 (64.2/99)
ResolveArchDefects...	'Quality DefectsRemaining' 34.8.	1
DetailDesigns...	On completion, 184.3 defects remain (120.9+63.4) i.e. 'Quality DefectsRemaining' 184.3. We infer that $184.3-34.8 = 149.5$ defects were injected.	$150/149.5 = 1.003$
InspectDesigns	Of the 184.3 defects remaining, 120.9 are detected. We now have Remaining=184.3; Detected=120.9.	.66
ResolveDesnDefects...	120.9 defects resolved in Design documents leaving $184.3-120.9=63.4$ remaining. We don't have enough information to know how many of those fixed originated in, and were fixed in, Architecture documents. We will assume the Method fixes Design defects only.	1
Code...	On completion, 312.8 defects remain (205.2+107.6) i.e. 'Quality DefectsRemaining' 312.8. We infer that $312.8-63.4=249.4$ were injected	1
InspectCode	Of the 312.8 defects remaining, 205.2 are detected. We now have Remaining=312.8; Detected=205.2..	.66 (205.2/312.8)
ResolveCodeDefects...	114.4 defects are resolved in code, leaving $409.3-114.4=294.9$ remaining. We don't know how many of these originated in Architectures and Designs and so we will assume the Method fixes code defects only. We also see that from table 4 after test the number is 304.2. 10.3 new errors were injected during fixing. So this Activity resulted in 114.4 resolved errors, 10.3 new ones injected.	$104.1/114.4=.91$.
UnitTestCode	Of the 112.5 defects remaining after unit test, 70.5 were detected	$70.5/112.5=.627$
ResolveCodeDefects...	70.7 defects resolved in code, leaving $112.5-70.5=42$ remaining. Again, we don't know how many of these originated earlier on. However after process test, 49.6 defects remained (30.1+19.5). So this Method caused 7.6 defects to be injected, ie effectively fixed $42-7.6$ corresponding to an Effectiveness of $36.4/42=.867$ (instead of 1 for all fixed). So after this Method 'DefectsRemaining'= 49.6.	$36.4/42=.867$
IntegrateAndTest	Of the 49.6 defects remaining, 30.1 were detected	.607
ResolveIntegrnDefects	30.1 defects are resolved, leaving a DefectsRemaining of 19.5. All fixed.	1

Table 8.17: State-based Activity effectiveness summary

Activity	As-is	To-be
Analysis	1.01	1.01
ArchitectureInspection	.276	.65
ArchitectureRework	1	1
DetailDesign	1.003	1.003
DesignInspection	.278	.66
ResolveDesignDefects	1	1
Code	1	1
CodeWalkthrough	.28	.66
WalkthroughRework	.91	.976
UnitTest	.63	.627
UnitTestRework	.95	.867
Integration	.67	.607
IntegrationRework	1	1

- There are some further issues around rework rates. Rework at the end of a process is more expensive because a larger number of documents must be changed. The cost will depend upon the origination points of defects. For example, if 100 defects are found in code, how many of these must be fixed in code only, how many in designs and how many in designs and architectures? Furthermore, if inspections are more effective, presumably a larger percentage of code defects will originate ‘upstream’ i.e. the rework rate will drop. It is possible that the rework rates used in the simulation are numbers obtained either from the literature or from previous experiments. If this is the case, these potentially represent hidden assumptions of the model.
- *Method* ‘ResolveCodeDefects ...’ after code walkthrough has relative effectiveness of .91. This figure seems low, possibly due to stochastic nature of simulation. But in a real-world situation it might indicate that the process is not fully understood. For example, perhaps the engineers were rushed and so less effective, indicating that some of the value ought to be attributed to *ContextModel*.
- The variations in numbers of errors injected, detection capability etc. arise as a consequence of the stochastic nature of the simulations. On data extracted from real projects, this variation would appear as a variation in *Activity* effectiveness. In such cases, it might be that the variation in values provide us with real information about what was going on during the project i.e. how *Context* affected the engineers’ ability to carry out the *Method*.
- Thinking in phases is potentially misleading, as most phases result in change to docu-

ments produced in prior phases.

The authors of the paper note that alternatives may be examined for the achievement of the suggested improvement i.e. changing error detection capability of inspections from 30 percent to 70 percent. Capturing the simulation in the *KiTe* framework provides a clearer understanding of what these alternatives might be. *Activity* effectiveness is achieved as a result of a base transformation defined by *Method* and *Technique* modified according to *ContextModel*. It is clear that either or both may be varied to achieve the desired value. I submit that that this transparency supports any business case made as a result of the simulation experiment, because the range of possible options becomes more obvious.

I also observe the need to truly understand what are the *Methods* being implemented and the source of the data being used. Some of the questions above may have straightforward answers. However, issues such as knowing origination points of errors, *Methods* with surprisingly low performance values and apparently contiguous tasks varying widely in cost would, in real projects, require attention. The use of a framework helps expose such issues.

8.2.8 Study 8: System dynamics simulation model

The next study concerns a system dynamics simulation model of a waterfall process. It involves a feasibility study based on work by Pfahl and Lebsanft [129] from the Fraunhofer Institute for Experimental Engineering (IESE) in Kaiserslautern, Germany. The cited paper describes a software simulation model developed by the Fraunhofer IESE for Siemens Corporate Technology (Siemens CT) and reports on results obtained from simulations. The purpose of the model was to demonstrate the impact of unstable software requirements on project performance and to analyse how much effort would be required to stabilise requirements such as to achieve the most cost-effective outcome.

The goals addressed by this study are 1.2.1.1. (traditional process), 1.2.2.3. (system dynamics simulation model), 1.2.3.1. (large-grained process), 1.2.4. (variations to base process), 1.2.5.1. (industry project), 1.2.6.3. (medium size project), 1.2.7.1. (CMM level 1), 1.2.8.2. (co-located project), 1.2.9.2. (upgrade project), 1.2.10.1 (standard goals).

Some key aspects of the environment are:

- Within one of Siemens CT business units, requirements for software projects are under the direct control of a system engineering group (*se*). This group solicits requirements from customers, and makes decisions about which parts of the solution are to be implemented in software and which in hardware. Software requirements are then passed to the software development group (*dev*) for implementation.

- Software project deliveries generally comprise three increments. The first implements base functionality and provides a prototype for customer feedback. The second implements important requirements, and the third completes all requirements, including customer-specific adaptations.
- There are three or four releases (I-cycles) during each increment. Generally, as a result of these releases, new requirements are received from the customer. These are planned for and included by *dev* in subsequent releases for the increment.
- In many cases, *se* chooses to change, replace or implement in hardware, software requirements that have already been implemented by *dev*. This often happens late in the project.
- The last item above is believed to be causing problems of project performance. Siemens CT are interested in knowing how much additional effort by the *se* group would be required to stabilise requirements such as to effect the most cost-effective outcome.

The simulation model developed by Fraunhofer IESE to provide the above information is based on the system dynamics paradigm. The ‘causal’ relationships identified as most relevant imply that increasing the number of already-implemented software requirements that are replaced during the project results in a longer project duration. Simulations imply that an increase in *se* effort from 10 to 42 person weeks (representing an increase in percentage total effort from 1.7 percent to 9.1 percent) results in an optimal solution, with a total effort decrease from 596 person weeks to 462 person weeks.

KiTe representation

I now work through the recreation of the above with the *KiTe* model. As our current aim is to show model feasibility, we need in the first instance to demonstrate that we can ‘plug in’ a set of *Activities* that results in the same outputs as the system dynamics model. The end result will be to show that a simulation based on system dynamics can be captured with *KiTe*, and to gain a greater understanding of what might be assumptions inherent in the two model types.

Activities I list the *Activities* that form the Process. For each *Activity*, I then identify its *Method* along with relative effectiveness values for the associated *Technique*. No details of the *Techniques* implemented are given and so I assume these remain constant between simulation runs. From the paper, the assumption is that all outcomes result from the *Method* and *Technique* only and so I assume the effect of *ContextModel* is negligible. I show the results for each of the six simulation runs described in the paper (see Table 3 of the paper).

The target process has three increments and each increment has a number of releases ([129] Fig. 1). For each release, there is a ‘gather requirements’ and ‘develop from requirements’ *Activity*. The process thus comprises one such cycle for each release. From Figs. 1 and 6 of the paper, I infer four releases for increments 1 and 2, and three for increment 3, making a total of 11 releases. From Fig. 6 of the paper, I also infer that a release spans 10 weeks.

The process comprises the following *Activities*.

- DefineForPrototype
- DevelopPrototype
- DefineFromPrototype
- DevelopRequirements
- DefineLastRequirements
- DevelopRemainingRequirements

Product The study focusses on software content and effort, and the quality focus is ‘correctness of requirements’. The latter must capture both requirements that are removed (for example, to be replaced with hardware) and requirements that are changed (presumably as a result of failure to capture functionality correctly). The first is a case of ‘they did the wrong thing’ i.e. were ineffective in initial capture. The second could imply the same thing, or could be a result of capturing the right thing, but doing it badly (‘they did it wrong’). The target study does not differentiate between the two cases (there is a single ‘weekly replacement factor’), but as the focus of the study is requirements that are replaced, removed and changed, I model the incorrect requirements as failure to capture the right thing. I can model this as a *Quality* attribute i.e. view as errors in requirements. However, I prefer to view the failure to correctly capture requirements as a lack of effectiveness in building *Content* and model as a *Content* attribute rather than a *Quality* one. I note that the *Perspectives* applied represent views on the attributes only and so the choice does not affect model outcomes. Cost is measured in ‘person weeks’. I thus include Content and Cost perspectives, and apply, for each *Product Partition* (Definition, Architecture, Design, Source, Integration, Packaged), the elements:

Content Total The total number of requirements collected or implemented. This includes removed requirements, replaced requirements, and requirements in final form.

Content Correct The number of collected or implemented requirements in final form i.e. requiring no change.

Cost ActualHours The time cost of collecting or implementing ‘Content Total’ requirements.

For the recreation, as in the original study, *Quality* elements are ignored, presumably because these remain constant across process comparisons. I assume there is no *Product* at process start. For this study, the software is finally delivered when all requirements have been correctly implemented. With the *Product* model above, this occurs when the ‘Content Correct’ value is equal to its maximum value i.e. the value targeted for ‘Content Total’.

From the target study ‘Reproduction of the reference mode’ (Fig. 6 in the paper), I infer that the total number of delivered requirements is 2,000 i.e. ‘Content Correct’ must reach this total.

Methods Each *Activity* above is based on a *Method*. The *Methods* are all different as regards effect on *Product*. For example, ‘DefineFromPrototype’ captures requirements based on a prototype and there is an expectation that the requirements will not be complete at the end of the *Activity*. For ‘DefineLastRequirements’, on the other hand, it is expected that requirements will be finalised. The *Methods* for increments A, B and C are:

- CaptureRequirementsForPrototype (A)
- ImplementPrototype (A)
- CaptureRequirementsFromPrototype (B)
- ImplementRequirements (B)
- CaptureFinalRequirements (C)
- ImplementFinalRequirements (C)

However, because of the feedback-based structure of a system dynamics model, the *Method*-related differences between the three increments is not captured. I thus implement two *Methods* only, ‘CaptureRequirements’ and ‘DevelopFromRequirements’. For the same reasons, i.e. it is not possible to distinguish between the three increments as regards the effect of contexts, I work with only two *Activities*, ‘Requirements’ and ‘Develop’. I note a limitation in the system dynamics approach.

From [129] Table 2 in the paper, I see that the number of requirements at project start is 1000. I can not understand how to use the ‘Initial requirements fraction for increment B/C’, but from Fig. 6, I infer ‘R(A)’ 1200, R(B) 500 and ‘R(C)’ 300. The ‘New requirements fraction’ in Table 2 of .15 implies that .15 of the requirements for an increment are received after the increment start. The values I will work with are :

R(A) 1200 (60 percent): R(A0) 1020 (51 percent); R(ANew) 180 (9 percent)

R(B) 500 (25 percent): R(B0) 420 (21 percent); R(BNew) 80 (4 percent)

R(C) 300 (15 percent): R(C0) 260 (13 percent); R(CNew) 40 (2 percent)

As Figure 5 in the paper indicates most new requirements in an increment are received during the first five weeks i.e. during the first release, I will allocate these to the second release for each increment (the base requirements for the increment are allocated to the first release).

I now need to decide how to handle the changed, replaced and removed requirements. From Table 3 in the paper I learn that the ‘actual average requirements replacement per week’ (AARR) is 0.73 percent for the baseline case. In section 5.1., I learn that “the number of replacing requirements per week is proportional to the number of requirements known at project start”. With an initial requirements count of 1000 and an AARR of .73 percent, I calculate 7.3 requirements replaced per week, 73 per release (10 weeks) and 730 over the course of the project. This seems consistent with Fig. 8.

The natural way to do this is to assign a non-unit effectiveness to the requirements gathering *Activities*. As I have no information to the contrary, I assume this effectiveness is the same for each increment and choose a value for ‘Content effectiveness’ that results in actual number of 2730 over 2000 total requirements. I thus work with a Content effectiveness value of $2000/2730 = .733$. I will implement this number for each release other than the last for each increment. For the last releases, I will implement a Content effectiveness of 1. The reason for this is that, at the end of each iteration, all requirements for the iteration are delivered and this means that the last releases must effectively deal with getting things right i.e. all recaptured requirements are correctly recaptured. Note that, if at project end I had requirement errors, a different *Method* would be required — perhaps one without the word ‘correctly’. For the ‘develop’ effectiveness values, I assign a value 1 as it is inferred in the paper that the ‘Develop’ *Activities* introduce no problems.

I now examine the ‘Cost ActualHours’ associated with each *Method*. For the ‘develop’ *Activity*, I require some idea of the baseline cost in person weeks to implement a requirement, in order that I can see a greater cost resulting when the number of requirements being replaced increases (more requirements are implemented). There is mention in section 5.1 in the paper that, because “each increment is different in nature, there is a dedicated level of productivity assigned to each level”. Table 2 contains ‘Nominal average productivity - functional unit per person week’ for the three increments. However, it isn’t clear what is a ‘functional unit’, or whether the productivity difference between iterations is due to the different number of requirements for each iteration (1200, 500, 300), rework, a different development method, or different contexts resulting in different effectiveness values.

For a first pass, I will assume the variation is due to requirements rework. This is taken care of already (difference between ‘Content Total’ and ‘Content Complete’). Thus I assume a ‘flat’ cost per requirement. From Table 3 in the paper, I see that a software development effort of 586 weeks was required to implement 2730 requirements and this gives a cost of .21465 person weeks per requirement.

However, if I carry out the same exercise for the other simulation runs, I find the cost per requirement varies.

- Case 1: 875 person weeks / 3830 requirements = .22846 person weeks per requirement.
- Case Baseline: 586 person weeks / 2730 requirements = .21465 person weeks per requirement.
- Case 3: 499 person weeks / 2400 requirements = .20792 person weeks per requirement.
- Case 5: 452 person weeks / 2140 requirements = .21121 person weeks per requirement.
- Case Optimal: 420 person weeks / 2080 requirements = .20192 person weeks per requirement.
- Case 6: 416 person weeks / 2060 requirements = .20194 person weeks per requirement.

It's not clear what is happening during the various runs to effect this difference. For the baseline case I work with a cost of .21465 person weeks per requirement, and consider this as equating to a baseline cost effectiveness of 1.

It remains to calculate the 'cost per requirement' for systems engineering baseline requirements gathering *Methods*. This doesn't seem to be so easy, because it's not clear why the assumption that more time taken (lower *Cost* effectiveness) will result in better results (higher *Content* effectiveness). Fig. 4 in the paper gives us a relationship between 'weekly replace factor' and systems engineering effort. But this really is a statement about the effectiveness of the requirements gathering exercise and would seem to be a model assumption that taking more time results in fewer requirements needing replaced. There could be many ways to achieve this result, for example, using more expert engineers, better tools, more planning.

However, as I am attempting to reproduce the results presented, I do not need to worry about what is the basis for the above assumption. I simply accept that the relationship holds and use the results of the relationship to calculate what was the *Content* and *Cost* effectiveness for the requirements gathering activity.

For the baseline case, the 'weeklyReplaceFactor' (see Fig. 4) is $\text{Min}(.05, 1/x)$ where $x = (2+\text{effort})^{\text{pow}1.7} = (2+10)^{\text{pow}1.7} = 17.694$. So 'weeklyReplaceFactor' = $\text{Min}(.05, .0565) = .05$. Applying this weekly increase to a base number of requirements 1000, over a period of 200 weeks yields a final number of requirements 1,729. This number, of course, corresponds to the 'AARR per week' of .73 (see Table 3). For the baseline case, total requirements is thus 1729 + 1000 (new).

For all simulations I will work from the 'AARR' for systems engineering, rather than calculate this from the equation, as my interest is in finding *Content* and *Cost* effectiveness values.

Baseline Simulation

Table 8.18: System dynamics Baseline - Incr A Activities

Activity	Characteristics	Effectiveness
Requirements	1020 .0037	.733, 1
Development	.25 .21465	1, 1
Requirements	1200 .0037	.733, 1
Development	.5 .21465	1, 1
Requirements	1200 .0037	.733, 1
Development	.75 .21465	1, 1
Requirements	1200 .0037	1, 1
Development	1 .21465	1, 1

The cost that is the baseline case is 10 person weeks for 2730 requirements i.e. a Cost effectiveness of 1 results in a cost of $10/2730 = .0037$ person weeks per requirement. I will work with this baseline value. For Content effectiveness I have $2000/2730 = .733$.

Table 8.19: System dynamics Baseline - Incr B Activities

Activity	Characteristics	Effectiveness
Requirements	1620 .0037	.733, 1
Development	.25 .21465	1, 1
Requirements	1700 .0037	.733, 1
Development	.5 .21465	1, 1
Requirements	1700 .0037	.733, 1
Development	.75 .21465	1, 1
Requirements	1700 .0037	1, 1
Development	1 .21465	1, 1

Summaries of baseline simulation values for increments A, B and C are presented in Tables 8.18, 8.19 and 8.20. Column 1 lists the *Activities*, column 2 the characteristic values, for example, ‘number of requirements gathered’, and ‘person hours per requirement’ and Column 3 lists the Content and Cost effectiveness for the *Activity*. I include the ‘person weeks per requirement’ value from which the *Content* and *Cost* effectiveness values are calculated. For reasons of simplicity and limited space, I describe *Activities* from now as simply ‘Requirements’ and ‘Develop’.

Result is Total req. 2728; SysEng effort 10.1 ActualHours; Development effort 585.6 ActualHours; Total effort 595.6 ActualHours.

Table 8.20: System dynamics Baseline - Incr C Activities

Activity	Characteristics	Effectiveness
Requirements	1960 .0037	.733, 1
Development	.34 .21465	1, 1
Requirements	2000 .0037	.733, 1
Development	.67 .21465	1, 1
Requirements	2000 .0037	1, 1
Development	1 .21465	1, 1

Case 1

We now adjust the effectiveness of the requirements gathering *Activities* to give results compliant with Table 3 in the paper.

For each case, I calculate the new systems engineering *Content* and *Cost* effectiveness values and create a new input process with the new numbers.

AARR is 1.83 and systems engineering effort is 5 person weeks.

Case 1 results in 3830 requirements in 5 person weeks i.e. 766 requirements per person week. Cost effectiveness relative to the baseline case is $766/273 = 2.806$.

For Content effectiveness we have $2000/3830 = .522$. This value is approximate due to the fact that it is applied to all releases other than the last in each increment, where I assume they ‘do the right thing right’. Experimentation found .47 to give the correct result.

Development cost is .22846 person weeks per requirement. This equates to a relative *Cost* effectiveness of $.21465/.22846 = .94$.

Results for increment A are shown in Table 8.21.

Table 8.21: System dynamics Case n1 - Increment A Activities

Activity	Characteristics	Effectiveness
Requirements	1020 .0037	2.806, .47
Development	.25 .21465	.939, 1
Requirements	1200 .0037	2.806, .47
Development	.5 .21465	.939, 1
Requirements	1200 .0037	2.806, .47
Development	.75 .21465	.939, 1
Requirements	1200 .0037	2.806, 1
Development	1 .21465	.939, 1

Result is Total req. 3825; SysEng effort 5.04 ActualHours; Development effort 871.2 Actu-

alHours; Total effort 876.24 ActualHours.

Case 3

Case 3 results in 2400 requirements in 15 person weeks i.e. 160 requirements per person week. Cost effectiveness relative to the baseline case is $160/273 = .586$.

For Content effectiveness we have $2000/2400 = .833$. Experimentation found .832 to give correct result.

Development cost is .20792 person weeks per requirement. This equates to a relative Efficiency of $.21465/.20792 = 1.032$.

Results for increment A are shown in Table 8.22.

Table 8.22: System dynamics Case n3 - Increment A Activities

Activity	Characteristics	Effectiveness
Requirements	1020 .0037	.586, .832
Development	.25 .21465	1.032, 1
Requirements	1200 .0037	.586, .832
Development	.5 .21465	1.032, 1
Requirements	1200 .0037	.586, .832
Development	.75 .21465	1.032, 1
Requirements	1200 .0037	.586, 1
Development	1 .21465	1.032, 1

Result is Total req. 2398.12; SysEng effort 15.14 ActualHours; Development effort 499.4 ActualHours; Total effort 514.54 ActualHours.

Case 5

Case 5 results in 2140 requirements in 30 person weeks i.e. 71.33 requirements per person week. Cost effectiveness relative to the baseline case is $71.33/273 = .2613$. Experimentation found .262 to give better result.

For Content effectiveness we have $2000/2140 = .935$. Experimentation found .9345 to give correct result.

Development cost is .21121 person weeks per requirement. This equates to a relative cost effectiveness of $.21465/.21121 = 1.016$.

Results for increment A are shown in Table 8.23.

Result is Total req. 2139.8; SysEng effort 30.22 ActualHours; Development effort 453.22 ActualHours; Total effort 483.44 ActualHours.

Table 8.23: System dynamics Case n5 - Increment A Activities

Activity	Characteristics	Effectiveness
Requirements	1020 .0037	.262, .935
Development	.25 .21465	1.016, 1
Requirements	1200 .0037	.262, .935
Development	.5 .21465	1.016, 1
Requirements	1200 .0037	.262, .935
Development	.75 .21465	1.016, 1
Requirements	1200 .0037	.262, 1
Development	1 .21465	1.016, 1

Case Optimal (based on total effort)

AARR is 0.08 and systems engineering effort is 42 person weeks.

Case Optimal results in 2080 requirements in 42 person weeks i.e. 49.52 requirements per person week. Cost effectiveness relative to the baseline case is $49.52/273 = .1814$.

For Content effectiveness we have $2000/2080 = .9615$.

Development cost is .20192 person weeks per requirement. This equates to a relative Cost effectiveness of $.21465/.20192 = 1.064$.

Results for increment A are shown in Table 8.24.

Table 8.24: System dynamics Case Optimal - Increment A Activities

Activity	Characteristics	Effectiveness
Requirements	1020 .0037	.1814, .9615
Development	.25 .21465	1.064, 1
Requirements	1200 .0037	.1814, .9615
Development	.5 .21465	1.064, 1
Requirements	1200 .0037	.1814, .9615
Development	.75 .21465	1.064, 1
Requirements	1200 .0037	.1814, 1
Development	1 .21465	1.064, 1

Result is Total req. 2080; SysEng effort 42.42 ActualHours; Development effort 420.43 ActualHours; Total effort 462.85 ActualHours.

Case 6

AARR is 0.06 and systems engineering effort is 50 person weeks.

Case 6 results in 2060 requirements in 50 person weeks i.e. 41.2 requirements per person week. Cost effectiveness relative to the baseline case is $41.2/273 = .151$.

For Content effectiveness we have $2000/2060 = .9709$.

Development cost is .20194 person weeks per requirement. This equates to a relative Cost effectiveness of $.21465/.20194 = 1.063$.

Results for increment A are shown in Table 8.25.

Table 8.25: System dynamics Case n6 - Increment A Activities

Activity	Characteristics	Effectiveness
Requirements	1020 .0037	.151, .9709
Development	.25 .21465	1.063, 1
Requirements	1200 .0037	.151, .9709
Development	.5 .21465	1.063, 1
Requirements	1200 .0037	.151, .9709
Development	.75 .21465	1.063, 1
Requirements	1200 .0037	.151, 1
Development	1 .21465	1.063, 1

Result is Total req. 2059.9; SysEng effort 50.47 ActualHours; Development effort 416.75 ActualHours; Total effort 467.22 ActualHours.

Discussion

Although I have found values that comply with the reported results i.e. I have successfully represented the study in *KiTe*, a couple of questions and observations arise:

- It's not clear what the relationship between 'weeklyReplaceFactor' and system engineering effort (Fig. 4) is based on. The Systems Engineering effectiveness values are an alternate way of defining this relationship. It would be interesting to know what is the underlying change in *Method* or *Context* that provides the different simulation values.
- The figure for development *Cost* effectiveness has been inferred by averaging across the process. I assumed no inherent difference between the three iterations. This may be incorrect and I may need to redo with different values for each iteration.
- I note that the development *Cost* effectiveness seems to vary according to the number of requirements and I am not really sure what is the mechanism that underlies this. Is the dip at 'n5' meaningful? Perhaps additional manpower causes a drop in effectiveness? Or there is a penalty for changing requirements i.e. a changed requirements costs more

than a new one. This highlights some assumptions of the study or some inadequacy in reporting study details.

- Because of the application of a system dynamics approach, it is not possible to differentiate between increments. However, from a *KiTe* perspective, the increments are different because, for example, each has different preconditions for requirements. I have exposed what is arguably a limitation of the system dynamics approach.

8.2.9 Study 9: Concurrent programming field study

This study re-examines data from an industrial field study by Parrish et. al. carried out to examine programmer productivity as team size varies [126]. Goals addressed are 1.2.1.1. (traditional process), 1.2.2.4. (quantitative study), 1.2.3.3. (small grained), 1.2.5.1. (industry project), 1.2.6.4. (small project), 1.2.7.1. (CMM level 1), 1.2.8.2. (co-located project), 1.2.10.1. (standard goals).

The original study showed productivity for teams working on common code to be much lower than for individuals. This result is in direct contrast with results reported for pair programming experiments. The study re-examines the data for team size two i.e. for two developers working concurrently on the same code module and aims to find out if the role-based protocol characteristic of pair programming, for example, use of the same computer, regular switching of roles, is the reason for such differing results. There were 48 modules with development teams of size 2. A *concurrency* metric is used — this is defined as the degree to which programmers reported working on the same module during the same day. The authors acknowledged that, although this measure does not perfectly measure the degree of *collaboration*, it is a necessary precursor and positively correlated with collaboration. Pairs were categorised as ‘high-concurrency’ or ‘low-concurrency’ and the number of unadjusted function points (UFPs) completed per unit of time measured.

Table 8.26: Productivity v. concurrency level (Parrish et. al.)

Concurrency level	Mean productivity	Standard deviation
Low	4.709	3.973
High	1.125	0.726

The contracted project was to rehost a legacy time accounting system to a distributed environment. The product had over 3,000 screens and approximately a million lines of code. The new system supports over 400 distributed users.

T-test results of productivity versus concurrency level are shown in Table 8.26.

Results showed that the mean productivity of individuals was about four times higher than that of pairs. The authors believe it is unlikely that the lack of productivity was due to duplicated or conflicting work because the developers worked at the same location and used a good version control system. They conclude that the role-based protocol of pair programming combats the natural productivity loss of collaborative efforts.

Table 8.27: Concurrency Product Model

Perspective	Partition	Attribute	Meaning
Content	Design	UFPs	# unadjusted function points
	Source	UFPs	# unadjusted function points
Cost	Source	TimeUnits	# 15 minute intervals

KiTe representation

As for other studies, I capture mean results and discuss the statistical nature of the study in Section 10.5.

The first task is to define an appropriate *Product* model. The study measures the number of unadjusted function points (UFPs) per hour and names this ‘Productivity’. I implement a *Product* model as in Table 8.27. I choose to separate ‘UFP’s and ‘TimeUnits’ because I expect that working with simple units will provide a *Product* model that will facilitate comparisons with other models and in any case the compound values are easily derived from the simple ones.

I next identify *Method* and *Technique* and again find some lack of clarity in the study description.

Method

Preconditions: There is no mention of what the developers are working from i.e. what are the inputs to *Method*. The study mentions that the UFPs were measured “from preliminary design information”. As the study relates to a large-scale, more traditional project, it is likely that full and detailed design documents are available, but we do not know this for sure. This is important information, because ‘coding from designs’ and ‘coding from requirements’ represent different *Methods* in *KiTe*.

Transformation: We do not know what is being created by the developers. Unit-tested code? Built and integrated modules? If the latter, we are dealing with integration of individual modules into a large system and the cost of integration may be high. This cost will

effectively be a ‘hidden cost’ if we treat this *Method* as one that produces *Source* only, and the result will be that the cost of producing *Source* appears to be very high.

Technique The factor that is the focus of the study, i.e. level of concurrency, I treat as *Technique*. The reasons for this as presented in Section 8.2.4 and relate to the possibility that individual engineer attributes, for example, ability to work with others, might affect the importance of this factor. I thus have two *Activities*, both with *Method* ‘CodeFromDesigns’ and with *Techniques* ‘LowConcurrency’ and ‘HighConcurrency’. The relative performance values for the two *Activities* are shown in Figure 8.15.

Activity	Product Source	
	UFP	TimeUnits
Code (CodeFromDesigns:LowConcurrency)	4.71	1.00
Code (CodeFromDesignsHighConcurrency)	1.13	1.00

Figure 8.15: Concurrency Technique relative performance

This Figure is a direct representation of the results in Table 8.26 from the paper.

I now capture context-related attributes. The *Engineer* attribute ‘Professionalism’ is mentioned and the authors believe this is held constant. The authors also report that the developers used a “fourth-generation tool, a modern relational database” and a development environment with “report generators, COTS libraries, database systems, and other new components”. I include these in the model for *Context*. As these are also held constant, relative effectiveness is the same for both low and high concurrency i.e. the authors believe the difference in performance is due only to the different *Techniques* i.e. to the different levels of concurrency. The models for *Engineer* and *Context* are presented in Table 8.28.

Discussion

As a result of trying to represent this study in *KiTe*, I uncovered some areas of uncertainty relating to *Method* and *Product* definition, because it is not clear what *Product Partition* is input to *Method* and what is delivered. This means I cannot represent the study without first making some assumptions. I also notice a possible problem when trying to define what is *Technique*. The authors claim that ‘concurrency’ is a reasonable measure for ‘collaboration’, as they are positively correlated and version control decreases the likelihood of conflict and duplication.

Table 8.28: Concurrency Engineer and Context Model

	Attribute	Measure
Engineer	Professionalism	No measure given
Context	FourthGenerationTool	
	RelationalDatabase	
	WindowsDevelopmentEnvironment	
	ReportGenerators	

But this is an unsubstantiated claim and it would seem likely that developers working on code at the same time will produce a different result than developers working on the same code at different times as the first would presumably force more communication. Although the two appear to be positively correlated, this might not be the case once *Engineer* attributes are taken into account. In any case, in *KiTe* we would handle these as two potentially different *Techniques*.

The authors report that programmers were “professional” but this term is not described further. It provides contextual information and so is important for study duplication. I include it in the model for *Engineer*, noting that any realistic *ContextModel* would be unlikely to be able to work with such a term, as it says little about the developers’ capability to work with *Product* and *Technique*. Some interesting thoughts about the context factors mentioned, for example, ‘FourthGenerationTool’, involve the realisation that these alone are not particularly useful when trying to determine how well developers work. In a *KiTe* system, such context factors matter only in as much as these support developers and the use of ‘modern’ and ‘new’ tools could in fact cause problems for developers unfamiliar with them. A more complete *ContextModel* would include some *Engineer* attributes describing familiarity with environment. Another possible *Engineer* attribute suggested by the study description is ‘familiarity with subject area’ because the fact that a contract situation is involved would tend to imply that developers might not be familiar with the subject area.

8.2.10 Study 10: Variations in XP process

In Section 8.2.3, I captured a ‘typical’ XP *process* in *KiTe* and illustrated how *Product* and *Engineer* might change during a single iteration. In this study, I consider some variations and illustrate how these might affect outcomes. For this study, I first vary some of the *Engineer* attribute values at the start of the iteration. I then consider the possible situation of failure to quickly resolve build defects and examine how this might affect choice of *Methods* and outcomes. Goals addressed are 1.2.1.2. (agile process), 1.2.3.3. (small grained), 1.2.4. (variations to base process) and 2.2. (compare process variations).

KiTe representation

For the first part of this study, I apply the same models for *Product*, *Engineer* and *Context*, and implement the same *Activities*, *Methods* and *Techniques*. However, I now assume that the *Engineer* ‘Technical Skills’ and ‘Product Area Knowledge’ are lower at iteration start than as for the illustration in Section 8.2.3.

For the second part of the study, I return to the decision to ‘wrap up’ build and build defect resolution into a ‘BuildAndTestAndFixDefects’ *Method*, and consider the situation where build defects are not resolved quickly. For this case, *Product*, *Engineer* and *Context* models remain unchanged, but now the ‘Integration’ *Activity* is separated into ‘Build’ and ‘ResolveDefects’ *Activities*, with associated *Methods* ‘BuildAndUnitTest’ and ‘FixBuildProblems’. These *Activities* iterate until the build is ‘clean’ i.e. we have a ‘local’ iteration. I illustrate this case with both sets of *Engineer* i.e. those with high and low ‘Technical Skills’ and ‘Product Area Knowledge’.

I reproduce Figure 8.10 from Section 8.2.3 below in Figure 8.16.

	Product Definition				Source				Integration				Packaged				Engineer													
	# Stories	Remaining Defects	Known Defects	PersonHours	# Stories	Infrastructure (1-10)	Unit Test	Remaining Defects	Known Defects	Complexity (1-10)	PersonHours	# Stories	Infrastructure (1-10)	Unit Test	Remaining Defects	Known Defects	Complexity (1-10)	PersonHours	# Stories	Infrastructure (1-10)	User Tests	Remaining Defects	Known Defects	Complexity (1-10)	PersonHours	SubjectAreaKnowledge (1-10)	ProductAreaKnowledge (1-10)	TechnicalSkills (1-10)	Satisfaction (1-10)	Confidence (1-10)
Start	3	2	2	1	2	3	2	3	2	3	3	2	3	2	3	2	3	1	2	3	2	3	2	3	0	5	1	5	6	6
PlanningGame	10	3	0	5	2	3	2	3	2	3	3	2	3	2	3	2	3	1	2	3	2	3	2	3	0	6	3	5	8	6
ReleasesMetaphor	7	1	0	5	2	3	2	3	2	3	3	2	3	2	3	2	3	1	2	3	2	3	2	3	0	6	5	5	8	6
PP	7	1	0	5	7	3	7	5	2	5	8	2	3	2	3	2	3	1	2	3	2	3	2	3	0	6	5	5	9	8
PPRefactor	7	1	0	5	7	2	7	5	2	3	9	2	3	2	3	2	3	1	2	3	2	3	2	3	0	6	5	5	9	8
PPRefactorCollective	7	1	0	5	6	2	7	5	2	2	9	2	3	2	3	2	3	1	2	3	2	3	2	3	0	6	6	5	9	8
CodeIntegrate	7	1	0	5	6	2	7	5	2	2	9	6	2	7	5	2	2	3	6	2	7	5	2	2	0	6	7	5	10	9
FunctionalTest	7	1	1	5	6	2	7	5	4	2	9	6	2	7	5	4	2	3	6	2	7	5	4	2	2	6	7	5	10	9
Finish	7	1	1	5	6	2	7	5	4	2	9	6	2	7	5	4	2	3	6	2	7	5	4	2	2	8	9	5	10	9

Figure 8.16: XP process iteration

Labels along the top represent the models for *Product* and *Engineer* and labels on the vertical axis represent XP Practices. The values in the table depict values for the model attributes after implementation of the Practices.

In Figure 8.17, I show a possible alternative outcome when the starting values for *Engineer* ‘SubjectAreaKnowledge’ and ‘TechnicalSkills’ are changed from ‘5’ to ‘1’ and ‘5’ to ‘2’,

	Product Definition				Source				Integration				Packaged				Engineer															
	# Stories	Remaining Defects	Known Defects	PersonHours	# Stories	Infrastructure (1-10)	UnitTest	Remaining Defects	Known Defects	Complexity (1-10)	PersonHours	# Stories	Infrastructure (1-10)	UnitTest	Remaining Defects	Known Defects	Complexity (1-10)	PersonHours	SubjectAreaKnowledge (1-10)	ProductKnowledge (1-10)	TechnicalSkills (1-10)	Satisfaction (1-10)	Confidence (1-10)									
Start	3	2	2	1	2	3	2	3	2	3	3	2	3	2	3	2	3	1	2	3	2	3	0	1	1	2	6	6				
PlanningGame	10	3	0	5	2	3	2	3	2	3	3	2	3	2	3	2	3	1	2	3	2	3	2	3	2	3	0	2	3	2	8	6
ReleasesMetaphor	7	1	0	5	2	3	2	3	2	3	3	2	3	2	3	2	3	1	2	3	2	3	2	3	0	2	5	2	8	6		
PP	7	1	0	5	7	8	7	9	2	9	8	2	3	2	3	2	3	1	2	3	2	3	2	3	0	2	5	2	9	8		
PPRefactor	7	1	0	5	7	7	7	9	2	8	9	2	3	2	3	2	3	1	2	3	2	3	2	3	0	2	5	2	9	8		
PPRefactorCollective	7	1	0	5	4	7	4	9	2	7	9	2	3	4	3	2	3	1	2	3	4	3	2	3	0	2	6	2	9	8		
CodeIntegrate	7	1	0	5	4	7	4	9	2	7	9	4	7	4	9	2	7	3	4	7	4	9	2	7	0	2	7	2	10	9		
FunctionalTest	7	1	1	5	4	7	4	9	5	7	9	4	7	4	9	5	7	3	4	7	4	9	5	7	2	2	7	2	10	9		
Finish	7	1	1	5	4	7	4	9	4	7	9	4	7	4	9	4	7	3	4	7	4	9	4	7	2	4	9	2	10	9		

Figure 8.17: XP - Engineers with low skill levels

respectively. In the new scenario, The result of the ‘PairProgramming’ Practice (row 4) is an increased ‘Infrastructure’ (from 3-8), ‘RemainingDefects’ (from 5-9) and ‘Complexity’ (from 5-9). ‘Refactoring’ and ‘Collective Ownership’ do not improve the situation, because of the low technical skills. These skills do not improve throughout the iteration, because there are no developers who are highly skilled to act as mentors. However, ‘SubjectAreaKnowledge’ has increased. The high ‘Infrastructure’ and code ‘Complexity’ are ‘invisible’, as ‘Stories’ have been implemented as required and defects found at ‘FunctionalTest’ are reasonable in number. The result of the iteration is a code base that is complex and likely to cause velocity to decrease in subsequent iterations. Engineers are unaware of any problem and so ‘Satisfaction’ and ‘Confidence’ are high.

I now consider the possibility that build defects are not resolved immediately. Such a situation may occur in a number of contexts. The XP team might comprise two persons only, one experienced and one inexperienced, and the more experienced of the two is sick. The team might be located in different locations and problems in the communications link may occur. To capture these situations in *KiTe*, it is helpful to decompose the ‘BuildAndUnitTestAndFixProblems’ *Method* into two *Methods*, ‘BuildAndUnitTest’ and ‘FixBuildProblems’. These are applied iteratively until build defects are resolved. This model enables us to examine where in the process possible bottlenecks may occur. For example, if our *ContextModel* considers attributes such as ‘experience’ and ‘communication’ to be of relevance, and our *Context* is ‘only one experienced engineer’ and ‘automated integration system’, it is likely that ‘BuildAn-

dUnitTest' will expose little risk but 'FixBuildProblems' is risky. On the other hand, if our *Context* is 'developers in different locations', 'communications channels of poor quality' and 'automated integration system', then 'BuildAndUnitTest' is seen to be of high risk due to the possibility of the remote build being unavailable.

Discussion

In the first case above, I illustrated an extreme scenario in order to both show that capture of such a scenario is straightforward and expose some possible problems with the XP process model. The attribute values were selected to serve these purposes and were not based on any formal *ContextModel*, rather were based on a subjective 'what if' reasoning.

In the second case also, I selected scenarios specifically to help illustrate how different *Context* and *Method* selection might be applied when modelling in *KiTe*.

The key idea from this Section is that the ability to capture *Methods* at different levels of granularity and different attributes for *Context*, *Engineer* and *Product* is necessary if we are to model the large range of possibilities for a *RealisedProcess*.

8.2.11 Study A-H: Miscellaneous process elements

In this Section, I show how some miscellaneous process elements would be represented in *KiTe*.

A: Developers have a discussion

Product is not changed and so *Method* accepts all *Product* states and effects no change to *Product*. *Technique* is 'Discussion' with *Technique CapabilitySpec* naming the relevant attributes, for example, 'Subject Area Knowledge', 'Product Knowledge', 'Java'.

The model for *Engineer* includes skill values for relevant attributes. These are increased according to *ContextModel*, for example, all values increase, with values for low skilled engineers increasing more.

B: Coding standards

Context includes an attribute relating to the existence of coding standards. *ContextModel* uses this as input when 'matching' *Engineer*, *Technique* and *Product CapabilitySpecs*. If *Technique* relates to coding, it is likely that the size of *Product* transformation due to *Technique* will be higher as a result of the coding support.

C: Add developers late to a project

The result depends upon the details of *ContextModel*. For example, it might be that *ContextModel* returns a higher value if the addition of the new engineers increases the general experience and skill level. It might return a lower value if the addition takes the number of engineers above some limit that is believed to be an upper limit for effective communications.

D: Developers happier and more confident doing XP

‘Happiness’ and ‘Confidence’ may be included as *Engineer* attributes, *Technique* ‘XP’ an attribute that denotes working in pairs and *ContextModel* include a matching that working in pairs makes people happier etc. A more sophisticated model might involve *Engineer* characteristics, for example, ‘likes working with people’ being included and *Technique Capability* ‘PairProgramming’ including ‘People work together’. This would produce a different result from *ContextModel* depending upon *Engineer* attribute value.

E: Parallel tasks

In a ‘real’ project, parallel tasks are always followed by some kind of merging procedure. For tasks that relate to different areas of the product, the procedure may be conceptual only. For example, outputs of ‘Create test plans’ and ‘Design’ may be merged simply by placing outputs in a target location. Possibilities include no conflicts (i.e. changes to different parts of the product that do not affect each other), no apparent conflicts, but problems because of couplings between the changed parts of the product and conflict i.e. the product has been changed in the same place by more than one task. It is the job of the merging procedure to resolve any problems.

One way to represent the above in *KiTe* is to remain true to real life and apply a ‘Merge’ *Activity* after the parallel *Activities* have been applied one after the other. *Technique* might be ‘PutInLocation’ or ‘Automated’ if no conflicts are expected or ‘ResolveConflicts’ if they are. For *Method* ‘MergeCode’, *Method* precondition would be the existence of *Source* and results are changed *Source*. As always, *ContextModel* ‘matches’ *Engineer* skills with the *Technique* requirements to modify the effectiveness of the base *Technique* i.e. how well the merging is carried out.

F: Advertise milestone release contents to open source community

In *KiTe*, each project is associated with a single *RealisedProcess* and each *RealisedProcess* is associated with a single *ProductIdentifier*. A delivery to any stakeholder implies some kind of

versioning i.e. each delivery is associated with a different *RealisedProcess*. A delivery is the final state for the *RealisedProcess*.

G: Project retrospective

Product is not changed and so *Method* accepts all *Product* states and effects no change to *Product*. *Technique* is ‘Retrospective’ with *Technique CapabilitySpec* including attributes such as ‘Brainstorming’ or ‘DiscussingProblems’.

ContextModel might represent a belief that talking about problems makes people happier. The model for *Engineer* includes, for example, ‘Satisfaction’. *ContextModel* increases this as a result of *Technique* ‘Brainstorming’ or ‘DiscussingProblems’.

H: Technology transfer

This is similar to A. *Product* is not changed and so *Method* accepts all *Product* states and effects no change to *Product*. *Technique* is ‘TechnologyTransfer’ with *Technique CapabilitySpec* naming the relevant attributes, for example, ‘Computer Telephony’.

The model for *Engineer* includes skill values for relevant attributes. These are increased according to *ContextModel*, for example, only values for low skilled engineers increase.

I: Disturb a task

RealisedProcess is defined as a state machine and the events that cause change to states are defined as ‘StartActivity’, ‘ChangeContext’ and ‘EndActivity’ (see Section 7.2.14). A *Method* is disturbed by ‘ChangeContext’ and the resulting state change illustrated in Figure 7.6.

8.3 Compare Processes and Process Models

8.3.1 Study 11: Developer collaboration

In Sections 8.2.4, 8.2.5, 8.2.6 and 8.2.9, I presented four studies involving pairs of developers creating code. Two studies [120, 169] present results that indicate that developers working collaboratively produce better quality code with very little loss in productivity. Another study [109] uses this result in a model for simulating XP projects. A fourth shows that collaboration is about one quarter as productive as solo programming and concludes that it is pair programming’s role-based protocol that is the cause of the good results [126]. However, the second study [120] produces good results for *collaboration* (not pair programming) and this is in direct contrast with results of the last study.

In this Section, I show how representation in *KiTe* provides a means of knowing when it is appropriate to compare studies and a mechanism for effecting comparison. In order to illustrate comparison, I first ‘fix’ some of the ambiguities and assumptions uncovered during the attempt to represent the studies (see Sections 8.2.4, 8.2.5, 8.2.6 and 8.2.9). I first focus on *Method* and illustrate the results of *Method* on *Product* for the four studies in Figure 8.18.

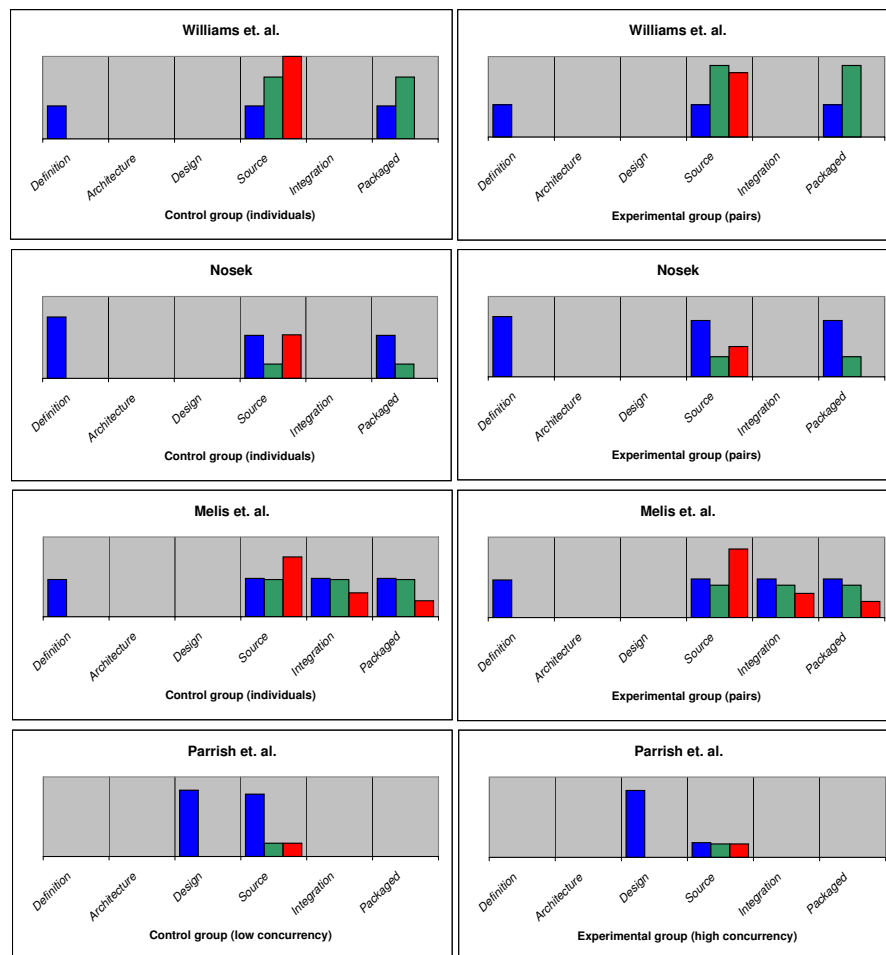


Figure 8.18: Case study overview

For each pair of diagrams, the ‘before’ state is depicted on the left and the ‘after’ state on the right. Each x-axis is divided into six — these are the *Product Partitions*. In each *Partition*, there are three bars. The left blue bar depicts the value of the *Content* attributes for the *Partition*, the middle green bar the value for the *Quality* attributes and the right red bar the values of the *Cost* attributes.

For the Williams study, I apply documented stories as *Method* inputs and code executables as deliverables. Integrations are not applicable and there is zero cost for packaging. For the Nosek study, I assume inputs are documented requirements and deliverables are stand-alone executable scripts. Again, integration is not applicable and there is zero cost for packaging.

For the Melis study, I assume a larger product with documented stories as inputs and delivery including integrated and executable code with non-zero cost for integrating and packaging. For the Parrish study, I assume developers worked from completed designs and deliverables were source modules, built locally only i.e. not formally integrated or packaged. I have omitted the y-axis values. These are values for *Product* attributes, but in the first instance I focus on the ‘shape’ of the *Methods* i.e. which *Product Partitions* are involved and so the actual *Product* attribute values are not helpful.

Comparing

The first interesting observation concerns the ‘shape’ of *Product* for each case. Cases 1 (Williams) and 2 (Nosek) have *Definitions* as inputs and result in changed *Source* and *Packaged Content*, with zero *Cost* applied to the last. Case 3 (Melis) also has *Definitions* as inputs but produces *Source*, *Integration* and *Packaged*, with some *Cost* applied to all three. The last case (Parrish) has *Designs* as inputs and results in *Source* deliverables only. I immediately understand that the *Methods* for the studies 3 and 4 are quite different to the *Methods* for the first two studies, and so studies 3 and 4 cannot be compared with studies 1 and 2 or with each other. Studies 1 and 2 are candidates for comparison and aggregation as the *Methods* for both are compatible. The key point is that, if *Methods* change *Product* in different ways, for example, one produces *Source* and another produces *Source*, *Integrations* and *Packaged*, it is not appropriate to directly compare studies based on them.

If *Methods* are compatible, I next consider *Product*. The Williams study measures completion times and successful test cases as percentages and the Nosek study measures ‘functionality’ and ‘readability’ on a number scale. The *Quality* aspect appears to be quite different in each case. Is it appropriate to consider *Quality* metric ‘readability’ to be synonymous with ‘percentage of test cases passed’ for the purpose of comparison? This would seem to be a hypothesis for study in its own right.

Once I have compatibility in both *Product* and *Method*, I then consider *ContextModel*. The *ContextModel* for a study represents what the researcher believes about context-related confounding factors i.e. context factors that might affect results. It is thus a vehicle for communicating with other researchers what was considered and what was accounted for. The Williams study holds constant *Engineer* capability and expertise and size of task i.e. works with a *ContextModel* that contains these factors only. The Nosek *ContextModel* includes and holds constant experience, motivation, size of task and familiarity with environment and tools. Experience levels in the two studies were different. Even if we were sure that the same *Method* applied in both studies, it is quite premature to assume that a similar performance outcome is the result of the *Method Technique* only, as I have no idea what might be the effects of capability

versus experience, fixed versus different motivation levels, etc.

To recap, if I am to aggregate and compare results from different studies, I must work with an abstraction that helps me know when it is appropriate to compare directly and when other, more sophisticated, mechanisms for evidence accumulation are indicated [131, 133]. Using the *KiTe* abstraction, I am able to check each possible problem area in a controlled and orderly way.

8.4 Discussion

In this Chapter, I have discussed the use of an approach called *argumentation* for collating and presenting evidence, and have introduced different kinds of evidence to support the thesis that the *KiTe* framework meets the objectives identified in Section 5.3.1. The attempt to provide this evidence has brought to light some interesting points.

The first relates to the importance of being able to select *Methods* at different granularities, in order that we might uncover possible assumptions when modelling a process. For example, in Section 8.2.3 I first applied a *Method* ‘BuildAndUnitTestAndFixProblems’, which appears to be compliant with descriptions in the XP literature, and then observed that greater insight into possible assumptions might be gained if the *Method* is decomposed into ‘BuildAndUnitTest’ and ‘FixBuildProblems’. A similar situation occurred in Section 8.2.1, where I noted that the many possibilities for defect resolution policy are simply not noticed if a large-grained ‘Design’ *Method* is applied and we need to be able to drill down more deeply if we are to understand what is really going on. The selection of appropriate *Methods* is not directly supported by the use of *KiTe* — rather the modeller must apply his or her own thought processes and real-life experience. However, as a result of the definition of a *KiTe Method*, the ability to work at any granularity is a feature of *KiTe* and so, once a particular process has been captured, experimentation with different granularities may be carried out.

The second point relates to the importance of selecting an appropriate model for *Product*. Choosing a model that is too restrictive results in study results that appear positive but, in fact, do not tell the whole story. For example, the XP researchers’ tendency to examine cost in terms of ‘person hours’, presumably because iterations are fixed in length, leads to a failure to more closely examine the integration task. Researchers assume that build defects are resolved ‘immediately’, but this is certainly not the case in many projects. For example, if the product is large and complex, or constituent parts developed in different timezones, it is possible that the build might remain ‘broken’ for several days. A consideration of a ‘Duration’ attribute might very well have brought such possibilities to light. Of course, the proponents of XP claim that, by ‘growing’ the product slowly and integrating frequently, the risk is minimal.

However, if we are to understand how to apply various Practices in different contexts, we must decouple the Practices and examine each separately and considering how a Practice might affect different kinds of product-related attributes is part of this examination. Again, *KiTe* does not directly support such efforts, but does so indirectly by forcing researchers to give consideration to attributes for *Product*.

9

Identifying Process Risks

In Chapter 5, I made a case for a return to the search for a theoretical model of the software process and in Chapter 7, I proposed a candidate framework, *KiTe*. One interesting side-effect of applying a model is that the model may be used to solve problems other than those for which it was created. An example of this is illustrated in Chapter 8, in which I showed that the attempt to capture various processes and process models resulted in exposure of many assumptions.

Another example of this kind of side-effect relates to the subject of process risk. In Section 2.3, I discussed the need to move discussion focus away from the level of ‘agile versus traditional’ and towards the characteristics of the various solution elements that make up any process. I suggested that such a focus might facilitate the identification of risk conditions inherent in the elements and support the building of process-specific risk profiles.

In this Chapter, I expand on this idea and show how a framework such as *KiTe* might be applied as an aid to risk identification. In the next Section, I overview the area of project risk and in Section 9.2, I show how *KiTe* is used to identify risk conditions inherent in an XP process.

9.1 Overview of Risk Management

The idea of managing project risks is not new. The Project Management Institute’s “Guide to the Project Management Body of Knowledge (PMBOK Guide)” [135] includes risk management as

one of nine key knowledge areas. The Guide defines risk management as “. . . the systematic process of identifying, analysing, and responding to project risk” with the aim of “. . . minimizing the probability and consequences of adverse events to project objectives.” Project risk has origins in the uncertainty that is present in all projects i.e. the possibility of occurrence of unplanned or uncertain events or conditions that have a negative effect on project outcomes. The PMBOK Guide cites six major risk processes, including Risk Identification, Analysis and Response Planning. Risk identification is to a large extent subjective in nature. The Guide suggests a number of techniques, for example, brainstorming and assumptions analysis, along with a list of risk categories appropriate to the application area, for example, technical, organisational, project management risks.

The Software Engineering Institute (SEI) supports a taxonomy-based approach to risk identification [26]. Possible risks are categorised as ‘Product Engineering’, ‘Development Environment’ and ‘Program Constraints’. Each category contains a list of factors and an associated questionnaire contains a numbered list of questions as guidance for risk elicitation. The SEI also approaches the issue of software project risk by proposing a construct for describing risks [56]. A software-dependent development effort is represented as a complex system with an n-dimensional space, with each dimension relating to a project characteristic that is agreed to be of relevance. Cited examples of characteristics include ‘program staff size’, ‘number of lines of code’, ‘requirements stability’ and ‘development model’. The project at any given time is represented as a fuzzy point in the space, and risk is viewed as a potential state-space transition from an acceptable state to an unacceptable one. A risk is thus expressed as a construct, the *Condition-Transition-Consequence (CTC) Construct* comprising a description of the initial state (the “condition”), the potential change to the system (the “transition”) and the potential final state (the “consequence”). One of the consequences of use of the CTC construct is a change in emphasis from consideration of ‘what might happen’ to ‘what are the system conditions that might equate to a risk’. A cited example is ‘the graphical user interface (GUI) must be coded using X Windows and we do not have expertise in X Windows’ (Condition); ‘there is a concern that the GUI code will be late and inefficient’ (Consequence). The emphasis is away from root cause analysis of possible impact and towards consideration of project current conditions coupled with concerns about these conditions.

Indirectly related research is provided by Curtis et. al. as a result of a field study of the software process for large systems [36]. The authors propose a behavioural model of the software development process and present the results of the study in terms of the model. The main findings of the study are that project problems are overwhelmingly caused by a small number of human-related factors. Curtis summarises these in a later paper as relating to lack of shared vision and domain knowledge, requirements uncertainty and issues of communication [34] and presents a case for a ‘behavioural model’ of the software development process that gives due

weight to the importance of the human element to project success.

The approaches presented above provide different kinds of guidance for identifying project risks. The PMI processes for risk identification include application of techniques, such as brainstorming, and use of tools, such as categories. This approach is applicable to all projects. The SEI approach involves examination of current project conditions for conditions that might contribute to unwanted project states. Identification is supported by a checklist, but this checklist is really aimed at large, traditional projects. For example, most of the questions under ‘code and unit test’ assume completed designs. The Curtis field study supported by other research [157] raises the possibility that a small number of factors create the greatest risk and exposes the danger in an over-emphasis on the technological aspects of a project. The implication is that risk identification would be better based on a more ‘behavioural’ process model.

The process modelling framework, *KiTe*, is an example of a behavioural model of the software development process in that both human and technological aspects are accounted for. As it represents a model of ‘how things are’, it also provides a framework for identifying project conditions that may be used to establish *CTC* risk constructs.

From a project perspective, risk management is concerned with the possibility of failing to meet project objectives. In *KiTe*, the project product-related objectives, for example specific values relating to quality and cost, are captured in the *GoalsBenchmark*. The consequence of any risk that eventuates equates to a failure of the delivered product to reach its expected values as defined in the *GoalsBenchmark*. Attribute values are achieved as a result of the *RealisedProcess* that is implemented i.e. to application of *KiTe Activities*. If the *GoalsBenchmark* changes during the project as a result of an agreement with stakeholders to, for example, lower quality expectations or include additional content, some existing risks may disappear or some new ones may appear.

I note that risk is also present if the meaning of a *Product* attribute is unclear, or the attribute is an inappropriate measure for the factor-of-interest. For reasons of practicality, I assume measurement risks are minimal.

I can thus use *KiTe* to identify project risks by first capturing a process in *KiTe* and then asking specific questions about the captured process by considering each framework component in turn. This is equivalent to identifying the *CTC conditions* for the elements. A captured process comprises a number of *Activities*, each of which comprises a *Method*, its associated *Technique* and a *ContextModel*, comprising *Engineer* and *Context*. I may consider *Activities* at any level of granularity i.e. I may view the whole process as a high-level *Activity*, then view each next-level *Activity*, and so on. When I perform a risk assessment, I first consider the process-level *Activity* and then each *Activity* at the next level. I may then further decompose *Activities* if required. For each *Activity* I ask the relevant questions. I may then wish to consider iterations, as these often serve to reduce risk factors. The questions are specific and reflect the

underlying *KiTe* model. For example, I first consider the precondition expectation for *Method* and ask if the required artifacts are available and of sufficient quality. Questions relating to *Technique* involve clarity of definition and availability and quality of required resources.

As illustration, I carry out an initial assessment of a project that will use a custom iterative process to create software that must run securely on a number of different platforms. The *Activity* is the *RealisedProcess* itself. The steps, with questions to be asked and possible risk conditions for the process-level evaluation, are:

1. *Method* precondition: *Are the relevant product artifacts available and of sufficient quality?* Input to the *RealisedProcess* is the *GoalsBenchmark*. Risk condition might be ‘security attributes not clearly defined or agreed’.
2. *Method*: *Are the tools required to transform inputs to outputs available and of sufficient quality?* ‘Unable to procure required development platforms and sufficient number of test rigs’.
3. *Technique*: *Is the Technique sufficiently defined and available and are required resources available and of sufficient quality?* ‘Process definition unavailable or unclear’.
4. *Engineers*: *Is there a skill match between engineers and relevant product artifacts and between engineers and techniques?* ‘No engineers with experience in security software or Linux’; ‘engineers new to process’.
5. *Contexts - certainty*: *Are engineers certain about what they are doing?* ‘Objectives and scope not clearly captured in *GoalsBenchmark*’; ‘communications issues on large, distributed project’.
6. *Contexts - support*: *Are engineers supported in what they are doing?* ‘Management not fully supportive because of pending restructure’; ‘poor support systems because relatively new company’.

I note that application to the *RealisedProcess* tends to expose lack of clarity in scope and objectives (*Method* Precondition), *RealisedProcess* (*Method* and *Technique*), manpower issues (*Engineer*), clarity of vision and potential communications issues (*Context*).

I now evaluate a single inspection *Activity* that involves reviewing design documents for design correctness and producing review reports. A company-wide on-line review system is used. A formal inspection *Technique* is to be used with four reviewers from the development group and an ‘inspection facilitator’ from a remote quality department.

1. *Method* precondition: *Are the relevant product artifacts available and of sufficient quality?* For the example, inputs are design documents and so risk conditions could be ‘designs unavailable’ and ‘designs not ready for inspection’.

2. *Method: Are the tools required to transform inputs to outputs available and of sufficient quality?* For the example, outputs are design review artifacts. Risk conditions could be ‘on-line review system unavailable’ and ‘on-line review system buggy’.
3. *Technique: Is the Technique sufficiently defined and available and are required resources available and of sufficient quality?* For the example, risks conditions could be ‘procedure definition unavailable’, ‘procedure definition unclear’, ‘inspector might not be available’, ‘inspector inexperienced’, ‘no office available’ and ‘office too small and dark’.
4. *Engineer: Is there a skill match between engineers and relevant product artifacts and between engineers and techniques?* For the example, the input *Product* artifacts are designs and so risk conditions could be ‘skills mismatch between reviewers and design language’ and ‘skills mismatch between reviewers and formal review procedure’.
5. *Context - certainty: Are engineers certain about what they are doing?* For the example, risk conditions could be ‘design documents incomplete or do not fully capture requirements’, ‘poor communications between reviewers’, ‘poor communications between reviewers and inspector’.
6. *Context - support: Are engineers supported in what they are doing?* For the example, risk conditions could be ‘management not supportive’ and ‘poor tool support’.

In the above risk identification example, the questions relating to *Engineer* and *Context* are specific in that they address skills, certainty and support. These questions reflect the use of a particular *ContextModel*, one that is based on the stated characteristics. I used this model because it encapsulates the most persistent ideas from the literature. Of course, once more sound evidence is accumulated, the questions will change to reflect the greater knowledge. For example, Acuna and Juristo’s ‘Human Competencies’ model [3] suggests personal characteristics, for example, ‘Privateness’, are key in matching *Engineer* and *Technique*, and use of such a model would result in the *Engineer* ‘matching’ question having a slightly different form. I also note that such a change in *ContextModel* would result in identification of different risk conditions. For example, in an XP process (see Section 9.2), an engineer with such a ‘Privateness’ characteristic would, according to Acuna and Juristo [3], have low ‘Negotiating Skills’ and might thus be ill-suited to a pair programming situation.

The list of questions based on *KiTe* elements is similar to the kinds of checklist described at the beginning of the Section, but based on current project conditions rather than possible risk outcomes. The SEI’s CTC approach is to identify project characteristics of relevance on a project-by-project basis. I submit that the use of *KiTe* as a framework for organising these characteristics may help risk engineers be more thorough and relevant in identification.

I note that the above approach may be applied for any software process. Initial capture of the process in *KiTe* requires the expertise of someone familiar with *KiTe* and perhaps some

‘real-world’ experience. However, once a process is captured, the questions are the same for any specific instance of the process i.e. the risk assessment process is generic. I also note that the questions relating to *Engineer* and *Context* reflect a particular *ContextModel* i.e. one that represents the belief that ‘skills’, ‘certainty’ and ‘support’ are key for a successful project outcome. As the industry accumulates evidence in the areas of *Engineer* and *Context* factors, an appropriate *ContextModel* may be inserted into the *KiTe* framework and a set of questions introduced that reflect the new understanding. The aim is, of course, that risk identification identifies those factors that are known to have greatest impact i.e. those represented by the *ContextModel*.

9.2 Risks in XP Process

We now analyse the XP process as captured in *KiTe* with a view to uncovering areas of potential risk in such a process [85]. Steps and questions are as described in Section 9. The first analysis views the whole XP process as an *Activity*. The associated *Method* requires a *GoalsBenchmark* as input and outputs a developed *Product*. This transformation is constrained to follow XP Practices i.e. the *Technique* is *XP*.

Project start

1. *Method precondition (DeliveryBenchmark): Are the relevant product artifacts available and of sufficient quality?* Objectives and scope are discussed during the *PlanningGame* and are generally not available at project start. This constitutes a risk condition, as there is no agreed ‘understanding’ between developer and customer about expectations.
2. *Method: Are the tools required to transform inputs to outputs available and of sufficient quality?* As objectives have not been defined, it is not possible to identify additional tools, for example special test rigs, that might be required.
3. *Technique: Is the Technique sufficiently defined and available and are required resources available and of sufficient quality?* There is a risk condition if there is any lack of clarity about any of the XP Practices. There is a risk condition if required resources are not understood or available, for example, the specific seating arrangements required for *Pair Programming*, common wall space, etc. The XP *Technique* requires the participation of a customer representative (external resource) as mitigation for the lack of pre-defined scope definition. There is a risk condition if there is any uncertainty about the authority, capability or availability of the customer representative.
4. *Engineer: Is there a skill match between engineers and relevant product artifacts and between engineers and techniques?* As scope and objectives have not been defined, it is

not possible to identify special engineer skills that might be required. It is possible that a skills mis-match will occur. A second risk condition would occur if engineers believe they understand XP but do not.

5. *Context - certainty: Are engineers certain about what they are doing?* There is uncertainty about scope and requirements.
6. *Context - support: Are engineers supported in what they are doing?* Risk conditions would be failure of management to understand the XP paradigm and lack of appropriate environments to support development.

Many of the above risk conditions are similar to those that might be identified for other kinds of process, for example, the lack of management support and development environments and lack of clarity about the process. Risk conditions specific to XP projects include a lack of up-front agreement about scope and objectives, the need for specific physical resources and the need for customer participation. Consequences would include lack of knowledge about required resources and developer skills, an inability to carry out the process and dependence upon the authority, capability and availability of the customer representative.

9.2.1 Single iteration

An iteration comprises a single ‘Planning’ *Activity*, several cycles of ‘PairProgramming’ and ‘Integration’ *Activities* and a single ‘CustomerTest’ *Activity*. We identify risk conditions for each *Activity*.

Planning

Method is ‘PlanningGame’. *Techniques* are ‘Metaphor’, ‘SmallReleases’ and ‘40 Hour Week’.

1. *Method* (PlanningGame) precondition: There is no precondition on *Product* and so no risk condition.
2. *Method: Method* outputs are informally captured Stories. Risks relate to availability and quality of tools required to produce these i.e. availability of notepads, pens and a suitable wall. Risk conditions would include ‘paperless office’, ‘no appropriate common wall area’ and ‘common wall area also used for other purposes’.
3. *Technique* (Metaphor, SmallReleases): ‘Metaphor’ is a somewhat abstract idea i.e. it is not clearly defined at all. However the customer representative is required to participate and create a suitable Metaphor. Risk conditions relate to his availability and understanding of the Metaphor concept and the required application. SmallReleases ensures that any decisions made in the form of Stories are small in number and unlikely to change

- i.e. *Method* outputs are certain. A perceived lack of progress may also emerge from *SmallReleases*.
4. *Engineer*: Risk conditions occur when developers do not have the required product-related skills (*Story capture*) or do not understand the relevance of the *Metaphor Technique*. Risk conditions include ‘developers unskilled at capturing the essence of *Stories*’ and ‘developers unfamiliar with the idea of shared vision’.
 5. *Context - certainty*: Certainty relates to shared vision and domain knowledge, requirements certainty and communications. As scope and objectives have not been defined, risk conditions are ‘customer is unavailable’, ‘customer does not have required knowledge’, ‘customer does not have product vision’, ‘customer and developers have different understanding of *Metaphor*’. However, any *Stories* agreed are fixed and certain.
 6. *Contexts - support*: Again, risk conditions occur when there is lack of management support.

Risk conditions inherent in the ‘*Planning*’ *Activity* relate to availability of appropriate space for *Stories* and capability of customer.

PairProgramming

Method is ‘*DesignCodeAndUnitTest*’. *Techniques* are ‘*PairProgramming*’, ‘*SimpleDesign*’, ‘*Metaphor*’, ‘*Refactor*’, ‘*CollectiveOwnership*’, ‘*CodingStandards*’, ‘*OnSiteCustomer*’ and ‘*TestFirstDesign*’.

1. *Method* (*DesignCodeAndUnitTest*) precondition: Risks relate to availability and quality of inputs (*Stories*). Risk conditions are ‘*Stories* and prioritisations not available where expected’ and ‘*Stories* or prioritisations unclear’.
2. *Method*: Risks relate to availability and quality of tools required to produce outputs. Outputs are unit test and source code. Risk conditions relate to availability and quality of development environment.
3. *Technique*: Risk conditions occur when any of the *Techniques* is not clearly defined or when required resources are unavailable. Examples of risk conditions include ‘physical layout not conducive to ‘*PairProgramming*’, ‘source control system doesn’t support free access to all code’, ‘coding standards unavailable or out of date’, ‘customer unavailable for problem solution’.
4. *Engineer*: Risk conditions occur when there is a skill mis-match between developers and *Product* and developers and *Technique*. If the *Techniques* ‘*PairProgramming*’ and ‘*SimpleDesign*’ are not carried out properly, the resulting code may implement *Stories*

incorrectly, be badly structured or contain excess defects. If ‘Refactor’ is badly executed, code will be badly structured. If at the same time ‘CollectiveOwnership’ either does not happen, or is carried out by unskilled developers, code will remain badly structured. The risk conditions occur if the developers are unskilled or under pressure. ‘OnSiteCustomer’ and ‘Metaphor’ mitigate incorrect implementation of Stories only if developers ask for help or understand ‘Metaphor’ concept.

5. *Context* - certainty: Risk conditions occur when developers have incorrect understanding (and so do not ask the customer) or when the customer is unavailable. Both conditions will result in incorrect implementation of Stories.
6. *Context* - support: Developers are supported by the presence of the customer representative. Risk conditions occur when he is not available.

Risk conditions inherent in the ‘PairProgramming’ *Activity* relate to physical layout and source control setup, unskilled developers, customer availability and understanding of Metaphor.

Integration

Method is ‘BuildAndUnitTestAndFixProblems’. *Method* outputs are integrated and packaged Stories. The *Techniques* are ‘DeveloperBuilds’, ‘ImmediateProblemFix’ and ‘IntegrateToPackaged’.

1. *Method* precondition: Risks relate to availability and quality of code and unit tests. Risk conditions are ‘Code untested or likely to break the build’.
2. *Method*: Outputs include integrated and packaged Stories. Risks relate to availability and quality of tools required to produce outputs i.e. build environment.
3. *Technique*: Risk conditions relate to availability and quality of required resources. As developers themselves perform the integration and resolve defects, the resources include a build setup that enables developers to easily work on- or off-line. The *IntegrateToPackaged Technique* captures the expectation that there is no separate task required to package integrated modules (see Section 8.2.3). There is a risk condition if this is not feasible, for example, if packaging results in a large overhead due to the need for copying of large data files, inclusion of user documents, stamping of user interfaces with build numbers, etc.
4. *Engineer*: Risk conditions occur when there is a skill mis-match with *Product* and *Techniques* i.e. if the developers are unskilled with the use of the build setup or unable to isolate and quickly resolve build problems.
5. *Context* - certainty: Risk conditions occur when there is any uncertainty. As developers carry out the whole procedure, there is little risk other than that relating to confidence in the build setup.

6. *Context* - support: A risk conditions occurs if developers are not supported in immediately resolving defects found. No specific risk.

Risk conditions inherent in the 'Integration' *Activity* relate to developers failing to immediately resolve problems i.e. to developer capability, the support provided by the build environment and the ability to package the application with minimal overhead.

CustomerTest

Method is *FunctionalTesting*.

1. *Method* precondition: Risks relate to availability and quality of integrated code.
2. *Method*: Risks relate to availability and quality of tools required to produce outputs. As the customer is responsible for test programs and rigs, these are out of the control of the development group. This is a high-risk situation.
3. *Technique*: Inapplicable.
4. *Engineer*: Inapplicable.
5. *Context* - certainty: Risk condition would involve some mis-communication between developers and customers i.e. if customer tests test something different to what is delivered.
6. *Context* - support: Inapplicable.

Risk conditions inherent in 'FunctionalTesting' relate to customer ability to implement high quality tests and understanding between customer and developers.

Iteration summary

Major risk conditions for an XP iteration include a physical setup not appropriate for an XP process, a dependence upon the authority, capability and availability of the customer representative, inadequate customer testing, developers who lack the appropriate skills or are under pressure and the infeasibility of packaging the application with minimal overhead. The consequences of these risks are poor communication (physical setup and customer availability), lack of clarity about scope and objectives (customer authority, capability and availability), untested product (customer testing), badly structured code containing many defects and incorrectly implemented Stories (developer skills and pressure) and slow progress (developer skills and packaging).

9.2.2 Process

XP is a highly iterative process and it is claimed that this provides mitigation for inherent risks. There are two kinds of iteration. The first is captured in the 'Continuous Integration' Practice

the second in the ‘Small Releases’ Practice. These Practices, along with ‘40-Hour Week’ appear in the ‘Process’ column of Table 8.6. We now examine these Practices with view to identifying new risk conditions.

40-Hour Week

This Practice really defines a resourcing policy and the claim is that policy application reduces pressure on developers. The risk condition of ‘developer pressure’ identified in Section 9.2.1 is removed.

Continuous Integration

This Practice involves developers integrating code every couple of hours. In our *KiTe* representation, the result is to effectively tightly-couple the ‘PairProgramming’ and ‘Integration’ *Activities*. Remembering that we are able to view *Activities* at any level of granularity, we introduce a ‘PairProgramAndIntegrate’ *Activity* i.e. we now have a ‘PlanningGame’ *Activity* followed by a number of ‘PairProgramAndIntegrate’ *Activities* and finally a single ‘CustomerTest’ *Activity*. The related *Method* is ‘DesignCodeUnitTestBuildAndUnitTestAndFixProblems’ and the *Techniques* now include an additional ‘Build2Hourly’ *Technique*.

1. *Method* precondition: As for *PairProgramming*.
2. *Method*: As for ‘PairProgramming’ and ‘Integration’.
3. *Technique*: As for ‘PairProgramming’ and ‘Integration’. Additional risks that occur as a result of the ‘Build2Hourly’ *Technique* relate to the resources required to effect the *Technique*. These include a build setup that supports a ‘build-and-test’ cycle of less than two hours. A risk condition occurs if the setup is too slow or if it is not possible to complete integration in the required timeframe.
4. *Engineer*: As for ‘PairProgramming’ and ‘Integration’ and including conditions relating to developers’ failure to understand the requirement.
5. *Context* - certainty: As for ‘PairProgramming’ and ‘Integration’.
6. *Context* - support: As for ‘PairProgramming’ and ‘Integration’.

Risk conditions inherent in the ‘ContinuousIntegration’ *Activity* relate to the ability to complete the integration in the required timeframe. Possible consequences of this risk are slow progress and increased defect rate due to the inability of developers to submit Stories immediately on completion.

Small Releases

This Practice captures a policy of defining only a small amount of work (number of Stories) at a time. One of the aims is to identify any misunderstandings and defects before these become embedded in the application. The single iteration of ‘Planning’, the *PairProgrammingAndIntegrate* cycle and ‘CustomerTest’ are repeated many times. No new risks are inherent in the ‘SmallReleases’ policy.

9.2.3 Discussion

Risk conditions identified above that are inherent in the application of an XP process are:

- Lack of up-front knowledge about required resources and developer skills.
- Dependence upon the authority, capability and availability of the customer representative.
- Physical setup not appropriate for an XP process.
- Developers who lack the appropriate skills.
- Inability to complete build-and-test in under two hours.
- The non-feasibility of packaging the application with minimal overhead.
- Inability of the customer to provide good tests.

Some of the above are acknowledged by Beck, who states XP is for “small-to-medium” projects and “it would not be possible to work in this style” if integration took a couple of hours [15]. However, with the latter, the implication is that a result of the Continuous Integration Practice combined with Refactoring is that only small numbers of modules will be built at any one time and so the 2-hour goal is achievable. This may work for many projects, but there are also many projects for whom the expectation is that delivery will be based on a full build, or where the packaging requirement is time-consuming (for example, user documents must be included, deliverables must be build-stamped). ‘Pure’ XP may not be a suitable approach for such projects and in fact some current research considers how to extend XP to address issues relevant to large systems [101].

10

Evaluation

I have proposed a model, *KiTe*, and claimed that the model is holistic and based on understanding. In this Chapter, I evaluate this dissertation in the following way. I first check *KiTe* for compliance with the properties defined in Chapter 6. These properties describe various process, model and real-world characteristics that are desirable and may be used as criteria against which to judge any candidate model. Satisfaction of the criteria provides some measure of confidence that the proposed model will successfully represent different kinds of process and real world situations and will solve some problems exhibited by existing process models. I then confirm that *KiTe* provides a solution that solves some of the problems I uncovered in the various attempts to model the software process in a flexible way (see Chapter 4).

I next overview the evidence presented in Chapter 8 in support of the ability of *KiTe* to meet two of the objectives defined in Section 5.3.1 and discuss the strengths and weaknesses of this evidence. I finally discuss the general approach taken in this dissertation and discuss strengths and limitations with the approach and with the model.

10.1 Evaluation against Criteria

In Section 6.2, I proposed some properties that any model should exhibit. I suggested that these properties might be used as criteria against which to judge a candidate model. The aim is to

provide some degree of confidence that the candidate model will address some of the issues with current processes and models and thus be more likely to succeed during the evidence gathering phase. In this Section, I evaluate the properties selected as criteria and evaluate *KiTe* in relation to the identified properties. I address each in turn and argue that *KiTe* exhibits the property.

Properties were identified from three different kinds of source. I first examined different kinds of process and identified characteristics that should be included in a holistic model. I then examined the problems with existing predictive models that rendered them inappropriate for general prediction and extracted suitable properties. I finally considered some real-life situations and again extracted some characteristics. From all of these, I extracted a set of properties that should be included in a suitable model (see Chapter 6). Because properties have been extracted from such different sources, I am confident that many of the characteristics that are key for holism and understanding have been addressed. However, the approach is ad-hoc and so it is possible that some essential properties have been missed.

I now evaluate *KiTe*'s ability to meet the criteria.

- P1** *Software processes only are represented. In particular, project management processes as defined in PMBOK are not included.* There is no *KiTe* component for planning, scheduling, risk management, scope management, etc. The *RealisedProcess* state space does not include states relating to these activities. Although *Product* may be extended to include attributes-of-interest to different researchers, the extension is to product-related attributes only, not management-related attributes.
- P2** *Product represents all descriptions of the artifacts that are delivered to the end customer. This includes problem descriptions, for example, requirements, and solution descriptions, for example, designs. Partitions are pre-defined and relate to all possible descriptions of a product. Definition attributes relate to problem descriptions, Architecture to the results of analysis and architectural decisions, Design to product designs, Source to code, product data, help files, text files, etc. Integration attributes relate to all build descriptions and Packaged to descriptions of the 'ready-to-deliver' product.*
- P3** *Product may be represented by a number of different measures. For example, representation might be 'lines of code' or 'number of requirements'. Product is abstracted as a model. The form of the model is defined but the actual attributes included are not. Different models for content, quality, cost, etc. may be used.*
- P4** *Product may be represented by more than one measure. For example, representation might include all of 'number of requirements', 'number of defects' and 'number of person hours'. This allows representation of, for example, both quality- and cost-related attributes. Product attributes are not constrained to any set. It is expected that Product will*

be viewed as a number of *Perspectives* and each provides a view on different kinds of attribute, for example, relating to quality or cost.

- P5** *Product attributes should be extensible in that new attributes can be included. Product is a model and, if new attributes are required, the model may be expanded to include these. Perspectives provide a convenient way of organising attributes in a meaningful way.*
- P6** *Processes may be represented at any level of granularity. For example, ‘create product’ or ‘carry out code inspection’. Method and Technique are abstracted as transformations on Product and start and end states are not constrained in any way. Any transformation on Product is thus legitimate.*
- P7** *Task definition is unambiguous. For example, for a task ‘design’, it is clear what the task changes and how it performs the change. Technique is a definition of Product transformation. Because all Partitions are involved, it is very specific about which attributes change. Method provides a mechanism for comparing Techniques as Techniques that apply to the same Method change Product in the same way and so may be compared.*
- P8** *A task may result in change to the humans carrying out the task and some tasks result in change to humans only. For example, developers become more satisfied as a result of participation in an XP project and design discussions do not change the product. People who change the product are represented in Engineer states which contain attributes that characterise the individual people, for example, skills. Activity is the component responsible for change to RealisedProcess state and changes all of Product, Engineer and Context. Activity may apply a Method that causes no change to Product.*
- P9** *Different beliefs about how human factors affect project outcomes may be represented. ContextModel uses information about characteristics of engineers, context, product and techniques to modify the changes to Product from Methods and Techniques. ContextModel is a model and its form may be selected according to the beliefs of the researcher or practitioner.*
- P10** *Some notion of ‘readiness for delivery’ is represented and its use optional. GoalsBenchmark captures the expected state of the product at process completion. The state machine that describes RealisedProcess has GoalsBenchmark states as final states, but it is not required in the state machine representation that these states be reached.*
- P11** *The model should account for product line processes, where a single conceptual product is changed by several projects and projects often deliver a product in more than one state. Product is modelled as a set of states in some global product state space, with all*

states particular to a specific *RealisedProcess* having a common *ProductIdentifier* that is unique to the *RealisedProcess*. This allows a product, for example, ‘MyWebApp’, to be changed by several projects. For example, one project may have states identified as ‘MyWebApp Prototype’ and another as ‘MyWebApp version 2.1’. I restrict a *KiTe Project* to have a single *RealisedProcess* and a *RealisedProcess* to have *Product* in a single state at any given time. This means that, for projects delivering, for example, both an early adopter version and a formal version of a product, it now becomes clear that the engineers involved in the real ‘single project’ are actually working on two *KiTe* projects in parallel. This relates to a change in *context* and corresponding change in how well the engineers are able to complete their assigned tasks. I submit that this provides an effective mirror on what is actually happening and exposes potentially ‘hidden’ situations.

P12 *Task parallelism should be supported.* The formal model specifies that *Product* may be in a single state at any point in time. This would indicate that parallel *Activities* may not occur. However, the model also specifies that *Product* state changes only on completion of *Activities*. If the *Activities* result in change to different parts of the product, for example, designs and integration tests, there is no conflict. As ‘Design’ *Activity* completes, *Product* state changes to reflect this and remains in this new state until some other *Activity*, for example, ‘IntegrationTests’ completes. If the parallel work affects the same part of the product, for example, developers changing a module at the same time, the *KiTe* representation must reflect what happens in the real world. As developer A completes, *Product* state changes to reflect this. When developer B then completes, someone must decide what is the ‘real’ state of *Product*, for example, by resolving conflicts when the module is committed to the source control system. This means that, until conflicts have been resolved, the source is in an ‘undefined’ state, both in the real world and in the *KiTe* representation. A possible *KiTe* representation would be to implement a *Technique* that accounted for the conflict situation (for example, ‘CodeWithConflict’). Such a *Technique* would concern a transformation on *Product* with lower cost-effectiveness than for a simple ‘Code’ *Technique*.

P13 *Model should be technology-independent.* *Technique* represents details about required technologies and is not constrained. Support for different technologies is provided via *CapabilitySpec* which allows specification of attributes working with the technology and matching of these with *Engineer* capabilities.

P14 *The model should represent the uncertain nature of the process by providing some way of capturing output ranges.* At present, *KiTe* does not possess this capability. I discuss this in Section 10.5.

10.2 Evaluation against Related Work

In this Section, I return to the studies presented in Chapter 4. These studies all relate in some way to the problem of comparing and synthesising processes. I compare the *KiTe* solution with those presented.

I first consider process frameworks, including Spiral, OPEN and RUP. I then discuss the approaches to process tailoring proposed by Basili and Rombach ('tailoring to project goals') and Boehm and Turner ('balancing agility and discipline'). I next address some simulation models for which claims of flexibility are made. I then discuss the experimental frameworks proposed by Kitchenham et. al., Basili et. al. and Williams et. al. and show how *KiTe* provides support for experimentation. I finally discuss how *KiTe* supports the representation of the various proposed models for people-related factors in the software process.

One observation relevant to this Section concerns the value of different abstractions of the software development process. Many abstractions are possible and useful in different circumstances. However, for an abstraction that will support general prediction, it is crucial that the right abstraction be applied. For many of the studies in this Section, the underlying abstraction did not support such generality, resulting in the limitations and assumptions described in Chapter 4.

10.2.1 Process frameworks

In Section 4.1, I introduced some frameworks whose purpose is to support project planners in the selection of appropriate process elements according to specific project contexts. These were the Spiral Model, the OPEN Process and the Rational Unified Process (RUP). All three frameworks provide a solution that includes both project planning and product creation. The planning component varies from mandatory inclusion of scope definition and risk management at each phase (Spiral) to the availability of support for some planning practices (OPEN and RUP). Limitations of these frameworks relate to the lack of guidance about which software processes to select and the lack of support for including human factors when selecting elements. Two of the frameworks, OPEN and RUP, also are specific to OO technologies and mandate these technologies at all points in the software process.

For the reasons given in Section 5.3.2, *KiTe* does not include direct support for project planning. For example, models for scope and risk planning are not included. However, *KiTe* provides a means of predicting the outcomes on *Product* of different choices of software process elements. As *Product* abstracts attributes of interest relating to *Content*, *Quality* and *Cost*, *KiTe* is effectively a tool to be used during planning and risk identification. I illustrate this by comparing the spiral and *KiTe* approaches. Spiral 'objectives', for example, 'functionality'

and ‘performance’, define what is the expected state of the product at time of delivery. Spiral ‘constraints’ are either product-related, for example, ‘cost’ and ‘schedule’, or context-related, for example, ‘developers are inexperienced’ or ‘test software is out-of-date’. From a *KiTe* perspective, there is no difference between product-related objectives and product-related constraints. Both describe what is the expected state of the product at time of delivery. These may be represented in a *KiTe GoalsBenchmark*. The task is now to find which choice of *Method* and *Technique* brings *Product* ‘closest to’ *GoalsBenchmark*. I observe that the results of some *Techniques* are affected by the context-related constraints (for example, a testing *Technique* may require specific test software). In *KiTe*, such constraints are represented as *Context* attributes. *Technique* outcomes are automatically modified depending upon the value of these attributes and the model for *ContextModel*. Thus, because relevant project factors are already represented in the *KiTe* model, some product-related risks are effectively accommodated in *KiTe*’s predictions.

The limitations of the studied frameworks included lack of guidance for process selection, lack of support for human factors and technology specificity. With *KiTe*, the restriction to a particular technology does not exist and the human-related aspect is provided by *ContextModel*. *KiTe Activities* may be selected according to *Product* state and change *Product* state in a defined way. This means that, at any point in a *RealisedProcess*, the selection of a ‘next’ *Activity* is constrained and its outcomes defined, thus providing a mechanism for self-guidance.

10.2.2 Process tailoring

In Section 4.2, I overviewed two approaches to tailoring the software process according to project environments. The first is an approach proposed by Basili and Rombach and involves improving a company process by selecting a specific goal for improvement, for example, defect numbers, and measuring the effects on this goal when various methods and tools are applied [13]. The second approach is proposed by Boehm and Turner and involves categorising a project along a traditional-agile scale as a means of selecting an appropriate kind of process for the project [22]. In both cases, the aim is to select a ‘best’ process according to project contexts.

In the Basili and Rombach approach, a major limitation was the constraining of the goal-setting to a single outcome. The example in the study related to ‘defect data’ with another possibility given as ‘customer satisfaction’. The use of a single goal is pragmatic. As the authors point out, the data required to support tailoring is not available and they view the approach as a way of acquiring this data by limiting the study to include a single outcome in the context of a single organisation. The *KiTe* solution encompasses this approach by allowing consideration of a single product-related outcome captured as an attribute of *Product*. It also provides a suitable framework for accumulation of study results because *Product* models may

be extended as required. However, *KiTe* goals are product-related and so, for representation in *KiTe*, an attribute such as ‘customer satisfaction’ would need to be operationalised into a number of product-related attributes, for example, quality- and cost-related. This may represent a limitation of *KiTe*.

The second limitation in this approach involved the variation in the list of factors that might affect outcomes. I noted that these covered a range of different kinds of factors, including some that represent product-related goals and some that represent engineer characteristics. In *KiTe*, factors such as ‘reliability requirements’ or ‘portability requirements’ are captured as ‘Quality’ attributes in *Product* and *GoalsBenchmark*. ‘Budget’ and ‘deadlines’ are also captured as *Product* and *GoalsBenchmark* attributes. ‘People factors’, for example, expertise and experience, are captured as attributes of *Engineer* i.e. in *Engineer CapabilitySpec*. Some factors noted in the authors’ list are captured as *Context* attributes that will be ‘matched’ by *ContextModel*, for example, ‘machine availability’. Some factors are not captured directly in *KiTe*. For example, ‘newness to the state of the art’ would be represented as one or more capabilities required by *Product* (say, a new technology, ‘x’), to be matched up with *Engineer* capability (say, ‘no experience with technology ‘x’’). This more specific approach enables *ContextModel* to provide results based on matching and also to increase the *Engineer* capability value as a result of working with technology ‘x’. ‘Programming languages’ are handled in the same way.

KiTe does not handle factors like ‘susceptibility to change’. This is a statement about *Product*, but not one that can be used to set goals or to match *Engineer* capabilities with required ones. It is a viewpoint that the likelihood of change to requirements belongs in the area of risk management. However, it is also a viewpoint that the inability to capture this as a product-related attribute exposes a limitation of *KiTe*.

Boehm and Turner examine the issues relating to ‘agile versus plan-driven’ process selection and believe that tailoring can be achieved by charting projects according to their ‘traditional versus agile’ characteristics and basing risk management strategies on the result. The approach is also a pragmatic one in that it provides an immediate way of approaching the problem of what kind of process to use. However, there is no further support in process selection other than the identification of risks.

In *KiTe*, the ‘home’ areas for traditional and agile projects would be represented in various models. For example, ‘criticality’ represents a product-related goal and would be modelled as a *Product* attribute and ‘culture’ as a *Context* attribute. The *KiTe* approach would also require the beliefs to be represented. For example, what exactly is the source of the belief that traditional processes are ‘better’ for criticality? Representation in *KiTe* would require statements about the effects of specific *Techniques* on a specific *Product* attribute, ‘criticality’.

10.2.3 Process modelling

In Section 4.3, I discussed a number of simulation models that exhibit characteristics of flexibility. In this Section, I show how *KiTe* addresses some of the limitations in these models.

Drappa and Ludewig created a simulation model for the purpose of project management training [46]. The model exhibited flexibility within certain limits (see Section 4.3.1). I noted that one limitation related to the lack of scalability of the model as a result of its rule-based nature with lack of any structure. I also noted that, because all model effects are captured as rules, beliefs about all aspects of the process are buried in the rule-base and effectively are hidden model assumptions. The abstraction of the process as a number of rules renders model extension impractical. For example, for the two contextual factors mentioned, ‘team size’ and ‘developer experience’, each rule relating some task to some outcome must be repeated four times to cover all context combinations. For three context factors, eight rules will be required, etc.

The *KiTe* abstraction mitigates the above problems. For example, instead of a large number of rules that represent an outcome value according to different combinations of input values, *KiTe* applies *Techniques*, each of which contains its effect on *Product* outcomes, and a *ContextModel* that ‘wraps up’ human-related factors according to a particular belief. New *Techniques* may be added to the system and will automatically be available for selection when *Product* is in one of the required precondition states. New contextual beliefs will be included in *ContextModel*. Scalability is addressed by the ability to work at any level of granularity when considering *Techniques* and by a suitable abstraction for *ContextModel*.

Lakey proposed a theoretical framework for project management (see Section 4.3.2). His framework comprises a number of building blocks and customisation is achieved by copying and renaming blocks and providing input values appropriate for specific projects [93]. Limitations included the pre-definition of both block function and key factors for affecting outcomes.

In *KiTe*, tasks are abstracted as *Methods* and these are not constrained in how they effect change to *Product*. The limitation of predefinition of block function is overcome. In Lakey’s model, the key factors include factors from all of process, product and project. Customisation is limited to selection of values that represent the environment for the project being modelled. In *KiTe*, factors are associated with different models. For example, process-related factors such as ‘defects injected’ are inherent in the *Technique* that represents a task and project-related factors, for example, ‘tool support’ and ‘skill level’ are represented in models of *Context* and *Engineer*. The relevant factors are defined by *ContextModel* and may be changed or extended by substitution of a new model.

In Section 4.3.3, I overviewed a model created by Munch for customising software development processes based on the concept of process patterns [115]. I note that Munch’s ‘char-

acterization vector' captures attributes whose values are changed as a result of transformations and his 'goal' is a restriction on these attribute values. The *KiTe Product* captures attributes that are changed as a result of application of *Methods* and *Techniques* and the *KiTe Goals-Benchmark* is a restriction on *Product* values. It would appear that 'characterization vector' is synonymous with *Product*, 'goal' with *GoalsBenchmark* and 'attribute transformation' with *Method*. Munch's model also has a 'quality model' that describes a cause-effect relation and this is consistent with the idea of the *KiTe Technique*.

Munch's model does not appear to include an abstraction of human-related factors and *KiTe* provides such an abstraction with *Engineer*, *Context* and *ContextModel*. I also noted a possible scalability problem due to capturing transformations as rules. *KiTe* addresses with a structuring of transformations into *Method* (families of *Technique*), *Technique* (may transform several attributes) and *ContextModel* (adds the human element).

10.2.4 Experimental frameworks

In Section 4.4, I described three frameworks for supporting researchers carrying out empirical studies on software processes. These were Kitchenham et. al.'s preliminary guidelines for researchers, Basili et. al.'s framework for families of experiments and Williams et. al.'s framework for XP studies.

Kitchenham et. al. note problems with defining contexts for studies and suggest some guidelines for recording contexts [90]. The *KiTe* framework demands an explicit capture during observational studies of both developer- and context-related information (in *Engineer* and *Context*). While this does not, at the present time, help us with establishing which factors are important, it does provide a means of recording which factors were taken into account and how these were believed to have affected outcomes. This transparency facilitates the comparison of studies because researchers can see at a glance which factors were considered for a study. Some evidence to support the notion that *KiTe* is useful for comparing studies is presented in Section 8.3.

Basili et. al. have the vision of a software engineering body of evidence to assist project managers in selecting processes for specific environments [14]. In Section 4.4, I identified the danger of introducing assumptions in the proposed framework. One problem is in the defining of 'Process' by the kind of task to be carried out, for example, 'Inspection' or 'Walkthrough'. I noted that an abstraction that allows a stricter definition of 'Process' is required. In *KiTe*, tasks are represented by *Method* and *Technique* and these are defined as transformations on *Product*. Such an abstraction means that tasks must be defined in a very specific way and the problem of introducing assumptions is minimised. The authors also note that much of software engineering experimentation involves tradeoffs, and so a common strategy is to assess, say, a technique in

comparison to a similar one. I note that *KiTe* allows such relative comparisons.

Williams et. al. propose a framework, XP-EF, for helping researchers establish compliance to an XP process. This framework does not support flexibility as it refers to XP processes only and predefines a set of relevant context factors. *KiTe* is not restricted to specific processes and different context factors may be included by changing *ContextModel*.

10.2.5 People factors

In Section 4.5, I presented three models that address the human-related aspects of software development. These were the ‘layered behavioural model’ of Curtis and associates [37], the ‘human competencies model’ of Acuna et.al. [3] and Acuna’s ‘team behaviour approach’ [4]. In *KiTe* these models represent ideas backed by empirical studies about the way in which human factors affect software process outcomes. Such studies would form the basis of a *ContextModel*. For example, for team formation, Acuna et. al. present models for ‘people’, ‘task’ and ‘team behaviour’. These map directly on to *KiTe Engineer*, *Technique* and *ContextModel*.

10.3 Evaluation against Research Objectives

In Section 5.3.2, I proposed that suitable objectives for a model of the software development process are that the model is able to capture any software process or process model, compare processes and process models and create a new process by combining elements from different processes. In Chapter 8, I presented evidence to support two of these objectives.

Bell reminds us that, to corroborate a theory, we must subject it to tests that could have shown it to be wrong. The aim is to increase the likelihood that the tests will reveal flaws [17]. The evidence presented in Chapter 8 provides some corroboration of the claim that the *KiTe* framework is successful in achieving the goals of process capture and comparison i.e. of two of the three stated objectives. The strengths of various pieces of evidence vary but, as a result of the transparent nature of the evidence map, can be easily assessed by interested parties. The objective of combining process elements has not yet been addressed i.e. there is, as yet, no corroborating evidence.

One key aspect of the provided evidence concerns the ability to represent both a Waterfall and an XP process in *KiTe* (see Sections 8.2.1 and 8.2.3). Such evidence is interesting because these two processes represent what are commonly believed to be incompatible approaches to developing software products. In Section 2.3, I suggested that discussions that categorise processes in a polarising way are unhelpful and that focus needs to be on understanding what are the key characteristics common to all software processes. The fact that I can represent both

in a common framework gives a clear indication that further investigation into the essential similarities and differences between apparently ‘opposite’ processes is possible.

The ability of *KiTe* to represent apparently different simulation modelling studies has also been demonstrated. In Sections 8.2.8, 8.2.7 and 8.2.5, I represented studies based on system dynamics, event-based and state-based models. Again, the ability to represent different kinds of models in a common framework is a first step towards greater understanding of the essential similarities and differences between models.

Another key aspect of the evidence is the ability to represent process variations (see Sections 8.2.2 and 8.2.10). In the study on Waterfall variations, I presented some alternatives for the coding phase in a waterfall model that arguably capture in a more realistic way what actually takes place. In the study on XP variations, I showed what might be the outcomes if less capable developers were involved. This evidence is important because it provides an indication that *KiTe* is able to describe slight changes in both task- and human-related elements of the process.

Related to the ability to represent variations is the ability to represent some miscellaneous processes that commonly occur in real life. Some evidence relating to such processes was presented in Section 8.2.11.

One important observation is that the act of representing processes and process models in *KiTe* gives rise to the exposure of many ambiguities and assumptions in the target processes and models. This was demonstrated in most of the studies provided as evidence. Before providing evidence to support the idea that use of *KiTe* facilitates comparison between processes, I had to first expose such ambiguities and assumptions and then choose an interpretation for each process. I was then able to show that some of the candidate processes could be directly compared and others could not (see Section 8.3.1).

Although an attempt was made to address ‘risky’ areas, it is clear that many areas of potential risk have not yet been tested. For example, attempts to represent processes have not included open source or distributed processes. Controlled experiments and qualitative studies have not been represented, nor have the various predictive cost and defect models.

A most important point is that all studies have been based on the existing literature. Although many studies reported ‘real’ projects, both large and small, it is likely that a ‘live’ study on a real project might display characteristics and suffer from problems not obvious from reported studies. This first seems more likely to be true for very large projects, where size contributes to greater complexity. However, it is possible that studies of relatively small projects would uncover issues, as people-related problems are equally likely although perhaps take a different form. Some evidence to support comparison exists but no evidence as yet exists to support combination.

As a side-effect resulting from evidence accumulation, I was able to make some interesting observations. When attempting to represent a ‘standard’ Waterfall process, I realised that the

process is not defined and depends upon the existence of local policies for defect resolution (see Section 8.2.1). When attempting to represent an XP process, I learned that many XP Practices represent instructions on how to carry out the design-and-code task (see Section 8.2.3).

A second side-effect is the observation that *KiTe* can be effectively used as a risk-identification tool. This was illustrated in Chapter 9. An interesting and practical area for future research is exposed.

I conclude by noting that I have successfully captured processes of varying kinds and granularities, models built using different technologies and a number of ad-hoc tasks that appear in real projects but are generally not included in process definitions and that, thus far, no counter-evidence has been discovered. This provides some confidence that a framework such as *KiTe* may be used to capture and compare different kinds of process and process models.

10.4 Evaluation of Approach

In Chapter 5, I identified a problem of a lack of a theoretical model of the software development process that will support the establishment of cause-and-effect relationships and provide a means of predicting process outcomes. I observed that such a model might not be possible and that substantial progress will depend on much research effort and collaboration.

In this dissertation, I have taken various approaches that serve to support such a research effort. These include:

- The provision of an initial framework allows immediate support for research with a long term goal of providing a theoretical model for prediction.
- Consideration of all groups currently involved in modelling for prediction has enabled me to understand limitations and strengths and made it more likely I have provided a solution that is useful for all processes and process models. The framework approach has provided an immediate way for researchers from different groups working with different paradigms and with different beliefs to represent their models in a common format with a view to better understanding essential similarities and differences.
- The use of an argumentation approach to capture evidence further supports the possibility of collaboration as strengths and weaknesses of evidence are transparent.

One possible limitation of the approach lies in the assumption that researchers in the field of software engineering are in a position, and have the desire to, collaborate towards the achievement of such a theoretical model. The field is characterised by fragmentation and a desire to produce results that are immediately useful to industry. Pleas for collaboration towards some

larger goal have been made by a number of researchers and have gone largely unheeded. The *KiTe* approach spawns several areas for different kinds of research, each of which may contribute to the long-term goal of flexible representation for the purpose of prediction. However, the field may not yet be ready for such a holistic approach.

10.5 Discussion

From Section 10.1, I note that *KiTe* meets all model criteria except that relating to the uncertain nature of the process. There are two aspects for discussion. The first relates to the infinite nature of the *KiTe* state space that is a result of the need to include ‘all possible values of all possible combinations of all possible attributes’ for products, engineers and contexts. The *KiTe* model describes an infinite state machine. *RealisedProcess* constrains the possible states to include those that describe the product, engineers and context that apply in the *RealisedProcess*. *Activities* further constrain the state space via *Method*, *Technique* and *ContextModel*. The result is still potentially infinite, in that the domain for an *Activity* includes a possibly infinite number of states. The second aspect for discussion relates to the deterministic nature of *KiTe*. Each *Activity* transforms each of its allowed start states to a specific end state depending upon *Method*, *Technique* and *ContextModel*. It is possible, perhaps likely, that such a model is overly simplistic for a human-intensive process. Evidence presented above did not include any that closely examined either the infinite or the deterministic aspect of the model. Possible alternatives for the nature of the elements that comprise *Activity* are sources of future study and are overviewed in Section 11.3.

From Section 10.2, I conclude that *KiTe* effectively encompasses and addresses the problems exhibited by existing solutions towards flexibility. One possible limitation is the inability to represent in *KiTe* a product attribute such as ‘susceptibility to change’. It is not clear at this stage whether such an attribute is best handled as a possible project risk or whether a *KiTe* limitation is exposed. In Section 10.2.2, I observed that contexts such as ‘newness to the state of the art’ can not directly be represented in *KiTe* but must be represented as specific *Product* and *Engineer* capabilities. This represents a benefit of the *KiTe* approach, in that what is meant by the ‘state of the art’ is explicitly defined thus removing possible sources of ambiguity.

From Section 10.3, I conclude that sufficient evidence exists to provide some confidence that a model for representing and comparing processes and process models is possible. The lack of any conflicting evidence supports such a confidence and indicates that *KiTe* represents a possible candidate model. I also note that some side-effects of a model such as *KiTe* include support for risk management and for exposure of ambiguities and assumptions. A side-effect of this ability to expose problems is the realisation that many problems might be mitigated if *KiTe*

is used to support the study design. For example, the need to define what is the *ContextModel* for the study leads to consideration of engineer- and context-related factors and of how engineer characteristics affect the way the engineers work with product and techniques. These represent possible confounding factors for experiments and factors that must be recorded for other kinds of study.

An interesting observation was made in Section 8.2.4, where I discussed some alternatives when representing ‘collaboration’. One way is to represent as a *Context*, for example, ‘developers work together’. *ContextModel* then applies the effect of ‘working together’. For this representation, all developers are equal in that there is no ‘matching’ taking place and individual developer characteristics are not relevant. A second possible way is to represent ‘collaboration’ as a *Technique*. In this case, *ContextModel* is able to directly match engineer capabilities (for example, ‘introverted’) with *Technique* requirements (for example, ‘extroversion’) to effect a more accurate result that depends upon which engineers are involved. This example represents a subtlety of the model that contributes to its usefulness and power.

In Section 10.4, I outlined some strengths of the approach taken in this dissertation. I also noted that the industry may not be in a position to effectively rise to the challenges of a collaborative approach to a model to serve as a basis for establishment of cause-and-effect relationships and prediction.

KiTe is not an implementation model and does not aim to represent all entities that would be required for model execution. For example, the model describes how *Activity* changes *Engineer* characteristics, but there is no element that acts as a datastore for engineer capabilities at different points in time i.e. the model does not allow for ‘managing’ engineers. It is likely that any implementation of *KiTe* for execution would include such an entity. A second issue relating to implementation involves the close-coupling between *ContextModel* and the models from which it sources its information i.e. *Context*, *Product*, *Engineer* and *Technique*. For example, if *ContextModel* requires knowledge about certain contexts and engineer characteristics, these must be provided. This may be a limitation as regards model usage.

A related discussion involves the form of the models that make up *KiTe*. For example, I note that, because *KiTe Methods* are defined as how *Product* is changed, it is possible to abstract at several levels and work in a hierarchical way. I also note that the form of *ContextModel* is not defined and a set of rules is one possible option. Such a choice would lead to the problems of scalability discussed in Section 4.3.1. However, the intention is that the models that make up *KiTe* represent theoretical abstractions rather than practical applications. It is appropriate to implement *ContextModel* as a rule base during early stages of exploration, but the expectation would be that the final model would represent a more theoretical abstraction. For example, as *ContextModel* represents engineer effectiveness, it is appropriate to identify an orthogonal set of ‘engineer effectiveness’ factors and all map all context factors on to this set. A suggested

set for preliminary exploration is the set ‘capability’, ‘certainty’ and ‘motivation’ (see Section 7.1). Such an abstraction means that we do not have to deal with large numbers of contexts but rather with a small set of factors. Indeed, it is likely that research from the social sciences might be leveraged for both identification of, and evidence to support, key factors for effectiveness. I note also that one outcome of the change in focus to effectiveness is that we can postulate that the factors under investigation are the same, regardless of project or task size. The problem of large numbers of factors has been ‘moved one step back’ i.e. once we understand that, for example, ‘certainty’ is key, we might then investigate what affects ‘certainty’ in large and small projects.

There is a possible limitation in the definition of *Technique* preconditions. For example, a *Technique* might define its preconditions in terms of both ‘LOC’ and ‘Function points’ but *Product* has only ‘LOC’. It appears that the precondition is not met. The solution is to expand the definition of precondition to include alternatives. For example, the precondition might be ‘size’ and may be actualised by either ‘Function points’ or ‘LOC’. The structuring of preconditions is another area for future research.

KiTe is a framework into which models are inserted. Each model represents a theory in its own right, and may be used to generate hypotheses to be tested in the normal way [151]. A current limitation is that the form of constituent models is not known and this means that predicting with *KiTe* suffers from the same problems of lack of support data as other predictive models. For example, there is no model for ‘matching’ of engineer capabilities with those required for working with a given product and techniques and so *ContextModel* represents beliefs. It is possible that the complexity of the software development process will render it impossible to successfully abstract all possible human-related factors into a *ContextModel* that successfully represents the human effects. It is also possible that the issue of measurement is insoluble and this would render it impossible to create valid models that are internally consistent and can be successfully manipulated.

Kitchenham et.al. remind us that, although researchers have a responsibility to “provide some preliminary validation of their results, they are not the best people to form objective, rigorous evaluations of their own technologies” [90]. The evaluation presented in this Section is subjective and therefore biased. I have, however, structured the discussion in an attempt to ensure that all relevant aspects have been considered. I have shown that *KiTe* may be used to represent very different kinds of process and process model and that it may be used to support research and risk identification. In the next Chapter, I identify areas for future research.

11

Conclusions

11.1 Summary

In this dissertation, I considered some viewpoints about the nature of software development and overviewed some of the processes used at the present time to produce software products. I suggested that the tendency to categorise these processes in a polarising way removes focus from the important task of understanding what are the essential characteristics of all software development processes. I then presented some of the work of researchers who model the software process with a view to understanding and predicting outcomes. I showed that existing models contain ambiguities and assumptions about process, product or contexts and this renders them unsuitable for representing and comparing processes in a general way. The motivation for the thesis is this lack of holism in existing processes and the inability of current models to represent any process and support comparison and combination of processes.

I then presented some perspectives on modelling and research data accumulation from fields other than software engineering. I concluded that existing models of the software development process are not based on cause-and-effect relationships and so cannot be used for predicting in a general way. I made a case for a theoretical model of the process and proposed that a candidate model should support the objectives of representation, comparison and combination of processes and process models. I examined existing processes, process models and some

real-life situations to derive some desirable model properties.

Using the desired properties as a basis for model structure, I introduced a candidate model, *KiTe*, and presented evidence to support the claim that *KiTe* meets the objectives of representation, comparison and combination. I discussed the use of *KiTe* for identifying process risks. Finally, I presented an evaluation of the dissertation by evaluating *KiTe* against properties and objectives and by discussing the approach taken.

In this Chapter, I summarise the contributions made and present some areas for future research.

11.2 Contributions

Despite many technological advances that support the production and maintenance of software-intensive products, “a body of evidence has not yet been built that enables a project manager to know with great confidence what software processes produce what product characteristics and under what conditions” [14]. The major contribution of this thesis is to identify the need for a holistic model of the software development process that will support researchers in their quest to accumulate such a body of evidence and to present a candidate modelling framework, *KiTe*. *KiTe* supports research by providing an abstraction that facilitates identification and representation of the various factors that affect process outcomes. As such, it represents a holistic and theoretical approach to software process modelling. Each framework element is a model that may be instantiated in the short term with models representing the beliefs of individual researchers and in the long term with models representing evidence-based theories.

Three contributions result from the understanding that the existence of a suitable framework gives rise to a number of unplanned research directions. The first concerns the use of *KiTe* for identifying process-specific risks. In Chapter 9.2, I presented a preliminary study on the use of *KiTe* to identify risks specific to XP processes. The use of *KiTe* in this way is immediately useful from an industry perspective. A second direction concerns the use of *KiTe* to help with understanding research results. In Section 8.2, I showed that, when representing studies in *KiTe*, many ambiguities and assumptions are exposed. This leads to a better understanding of study results. A related direction involves the possibility of using *KiTe* to help mitigate the introduction of such ambiguities and assumptions when designing new experiments. I discuss these directions in the next Section.

Another major contribution of this dissertation is the identification of the various research groups that model the software development process to predict outcomes and the understanding of how these groups differ in approach and what are the limitations inherent in the work of each. The contribution also includes a realisation that the narrow approach taken by each of

the groups is a symptom of lack of real understanding and is the basis for the case for a more holistic approach.

A final contribution is the establishment of an approach for developing and evaluating models that claim to describe systems in a holistic and explanatory way. The strategy is to first establish model objectives and identify a comprehensive range of example systems to be described by the model. In this dissertation, objectives were identified as ‘capture’, ‘compare’ and ‘combine’ software process elements and example systems included various kinds of process and process model. The next step is to examine the characteristics of, and problems with, the example systems to help identify key model properties. These provide some basis for establishment of a suitable model structure and may be used as criteria against which to evaluate candidate models in a preliminary evaluation step. Suitable properties for *KiTe* were sourced from process characteristics, process model limitations and real-world examples. Next, the ability of the candidate model to satisfy objectives is established by accumulating a portfolio of different kinds of evidence relating to the example systems. In this dissertation, evidence included representing and comparing various processes and process models. Finally, evaluation of a candidate model is performed by evaluating both the model’s compliance with property-based criteria and its ability to meet stated objectives.

11.3 Future Work

Because this thesis presents a modelling framework comprising models representing different aspects of the software development process, there are many possible areas for future investigation. These mostly involve attempts to acquire further evidence to support the claims for *KiTe*, as such activity will inevitably expose limitations and inconsistencies in the current model. One approach would be to effect an ad-hoc approach to the accumulation of further evidence. However, because the required evidence is partitioned into a small number of goals, and evidence in each partition potentially provides a different kind of benefit to modellers, I suggest that future work is best organised as a number of *KiTe* research programs, each with its specific goals. I overview some possible programs below.

11.3.1 Model foundation

The current *KiTe* model was presented formally in Section 7. If this model, or another like it, is to be used as a basis for hypotheses, it is necessary that it is sound and consistent. Model checking activity is indicated and some work in this area is under discussion at the current time.

In Section 10.5, I discussed the possible problem of a deterministic model to represent the software process with all its complexity. Once the model elements are defined the outcome is

known. The majority of the models presented in Chapter 4 manage the uncertainty inherent in real-world projects by applying some statistical mechanism. For example, a common approach is to use a probability distribution from which values for some input variables are chosen at random. The *KiTe* elements that affect transformation details, for example, *Technique* and *ContextModel* are defined as causing definitive change. This does not preclude the possibility of values being selected from a distribution, or indeed, being ‘fuzzy’ in the first place. Another possible alternative is to treat *Technique* as a ‘fuzzy’ functional mapping and the effects of *ContextModel* as fuzzy values. The outcome for any product attribute would then also be fuzzy in nature. A third possibility for future research would examine the possibility of modelling *Activity* as a Bayesian belief net [48]. According to Fenton, this approach deals with uncertainty, incomplete information and diversity of evidence. It might, however, involve ‘collapsing’ of *Technique* and *ContextModel* and so might be useful more as a possible simulation implementation than as model definition.

11.3.2 Process representation

A second possible program involves further representation of processes. In this thesis I successfully represented a number of different kinds of processes, for example, Waterfall and XP, and made a case for the inability to fully represent others, for example, Spiral. However, all case studies were from the literature and many assumptions had to be made. Focus must move to industry, and some ‘live’ projects, both large and small, studied. This will mean that questions can be asked in real time when any aspects of the process are unclear, and the need for making assumptions removed. The result should be either the provision of further evidence or the unmasking of model limitations and inconsistencies.

In addition to ‘live’ projects, a fruitful area for research is the representation of different kinds of process. For example, the *Cleanroom* approach to software development is viewed by some as “...not a strict methodology but a philosophical approach that guides the selection of practices” [40]. It would be interesting to know if Cleanroom is easily represented in *KiTe*. Other potentially fruitful processes include open source projects and Web-based virtual workspaces.

11.3.3 Process risk

It is claimed in this thesis that *KiTe* provides a framework to support risk identification. We have a powerful means of both supporting this claim and providing an immediate benefit to industry at the same time. I suggest a collaboration with industry projects that have strong risk management practices. Researcher and project manager separately identify risks, the project

manager using his normal approach and the researcher using *KiTē*, and then the two compare results. If the *KiTē* approach correctly identifies risks not identified by project management, we have some further supportive evidence. Risks identified only by project management may signal some possible need to modify the *KiTē* framework or may contribute to our understanding of one of the component models. For example, if the researcher is working with a *ContextModel* that includes aspects of uncertainty and support, and project management identifies a risk ‘the company is about to be bought over and developers may become demotivated’, there is an opportunity to rethink the *ContextModel* for possible inclusion of some ‘motivation’ factor. This approach of ‘comparing identified risks’ may be a powerful way for the research community to investigate the problem of developer efficacy whilst at the same time providing real benefits to industry.

11.3.4 Assumptions

The activity of representing different kinds of process models resulted in exposure of many assumptions made by modellers (see Section 8.2). These assumptions related to policies about what parts of the product are being changed, equivalence of product measures and the way that human factors influence results. I suggest that useful future work in this area would be to investigate ways of propagating this information within the various research groups. Until assumptions are understood and removed, it will not be possible to fulfil Basili et. al.’s vision of ‘families of experiments’ [14] and so not possible to use these models to progress software engineering research by providing empirical data.

11.3.5 Evidence

Yet another area for research is the comparing of different studies in order to find if results may be combined in some way. I presented one example of this in Section 8.3.1, where I attempted to examine three studies frequently cited as ‘pair programming’ studies and found that *Techniques*, *Product* and *ContextModel* for the studies varied so much that any kind of comparison or results accumulation was not possible. One potentially interesting subject for this research is inspections, as there are several different kinds of study available on this topic. A related area is that of support for formal experimentation, by using the *KiTē* models to capture experimental environments. For example, in *KiTē*, the focus is on identifying how contexts change peoples’ ability to carry out tasks. This means that, rather than identifying a myriad of factors that might affect outcomes, focus is on identifying how these factors affect humans and in turn how characteristics of humans affect outcomes. This change in focus has already been suggested by Curtis et. al. [37] (see Section 4.5). I suggest that *KiTē* may provide a suitable

framework for further research of this kind, as *Activities* can be defined with any granularity and experiments may thus focus on very small tasks carried out under many different circumstances. The long-term goal would be to spawn and examine hypotheses relating to the use of a single *ContextModel* for *Activities* of all kinds and granularity.

11.3.6 Product model

In Section 10.5, I noted a possible limitation in the current definition of *Technique* preconditions. The problem is one of understanding when product-related attributes are ‘equivalent’. For example, if a *Technique* expects as input ‘number of requirements’, is it valid to apply the *Technique* to a *Product* model with ‘number of stories’? This suggests a possible research program based on *KiTe* to further investigate such ‘families’ of *Product* attributes.

11.3.7 Process customisation

Another research opportunity concerns the use of *KiTe* for customising processes. In theory, *KiTe* supports this. However, as yet there is no evidence to support combination of process elements, i.e. the third objective for *KiTe* has not been tested, and this is key to customisation. In addition, attempts at customisation based on *KiTe* may raise issues and points for further investigation.

11.3.8 Predictive tool

A final interesting activity would be to commence implementation of the framework with the aim of understanding how construction of such a complex system might be supported in an open source environment. For example, stores of *Method* and *Technique* must be accumulated, each supported with existing evidence that provides the basis for the transformation size. *ContextModels* must be proposed and ‘evidence’ and ‘counterexamples’ uncovered. In the spirit of ‘evidence-based software engineering’, where large quantities of different kinds of weak evidence may provide confidence, it is possible that polling for such evidence via the web would obtain a good result.

11.4 And Finally

There are several interesting areas for future work based on the idea of a theoretical model for the software development process in general and *KiTe* in particular. Some of the programs suggested above aim to provide immediate benefits whilst improving the model in parallel. An

example, is the use of the model for risk identification. Others have less immediate practical use but are necessary if the model is to be used as a basis for theoretical studies, for example, the introduction of ‘fuzziness’ into the model. Kitchenham et. al., when discussing the need for a framework for validating software measurements [89], remind us that “A full, practical framework is an ambitious goal that requires input from practitioners and the research community”. This statement applies also to a framework for the software development process.

This thesis presents a framework that comprises a first step towards an ambitious goal. It represents an acknowledgement that there is insufficient data at the present time for process synthesis and prediction and provides a mechanism to support accumulation of such data by supporting comparison of processes and process elements. The thesis acknowledges a long term goal that is to provide a management tool for a software engineering ‘body-of-knowledge’ by providing a mechanism for comparing and predicting process outcomes based on evidence and where all evidence is transparent.



Glossary

For this document, the following terms have meanings as defined below.

Alphabet “An *alphabet* is a finite set of symbols.” [62].

Argumentation “...an approach which can be used for describing how evidence satisfies requirements and objectives” [160].

Capability [3].

Chief programmer A centralised organisation for programming teams, designed by Mills and Baker, that placed primary responsibility for design, programming, testing and installation on a single individual, the ‘chief programmer’ [33].

Construct validity “...the extent to which the variables successfully measure the theoretical constructs in the hypothesis.” [14].

Context The set of factors that affect how well *engineers* are able to carry out *tasks*.

Correctness “...the degree to which a system or component is free from faults in its specification, design, and implementation” [71].

Correlation study “Usually synonymous with *nonexperimental* or observational study; a study that simply observes the size and direction of a relationship among variables” [149] quoting Shadish et. al.

Defect A generic term to mean any one of *error*, *fault* or *failure* [13].

Design entity “An element (component) of a design that is structurally and functionally distinct from other elements and that is separately named and referenced” [73].

Effectiveness A measure of how well *engineers* complete *tasks*.

Efficiency A measure of outputs (change to a *product* from *engineers* completing a *task*) to inputs (some measure of time or cost).

Egoless team A decentralised team structure proposed by Weinberg [161] in which different team members take responsibility for those project tasks that amatch their skills [33].

Engineer Any individual involved in changing any artifacts that describe some aspect of a *product*.

Engineering “The application of scientific and mathematical principles to practical ends such as the design, manufacture, and operation of efficient and economical structures, machines, processes, and systems” [63]. “Engineering applies scientific and technical knowledge to solve human problems. Engineers use imagination, judgment, reasoning and experience to apply science, technology, mathematics, and practical experience. The result is the design, production, and operation of useful objects or processes” [163]. “...the systematic application of scientific knowledge in creating and building cost-effective solutions to practical problems” [43].

Entity attribute “A named characteristic or property of a design entity. It provides a statement of fact about the entity” [73].

Error “...defects in the human thought process made while trying to understand given information, to solve problems, or to use methods and tools” [13].

Evidence “A thing or things helpful in forming a conclusion or judgement” [64].

Experiment “A study in which an intervention is deliberately introduced to observe its effects” [149] quoting Shadish et. al.

External validity “...defines the extent to which the conclusions from the experimental context can be generalized to the context specified in the research hypothesis.” [14].

Failure “...the departures of the software system from software requirements (or intended use). A particular *failure* may be caused by several *faults* together; a particular *failure* may be caused by different *faults* alternatively; some *faults* may never cause a *failure* (difference between reliability and correctness)” [13].

Fault “...the concrete manifestation of *errors* within the software. One *error* may cause several *faults*; various *errors* may cause identical *faults*” [13].

Finite automaton “A *finite automaton* consists of a finite set of states and a set of transitions from state to state that occur on input symbols chosen from an *alphabet* σ . For each input symbol there is exactly one transition out of each state (possible back to the state itself). One state, usually denoted q_0 , is the initial state, in which the automaton starts. Some states are designated as final or accepting states.” [62].

GQM Goal/Question/Metric represents an approach to assist researchers in planning and categorising empirical studies. In this approach, researchers identify the object of study (for example, a process or product), the purpose of the experiment (for example, evaluation, prediction, etc.), the focus i.e. the aspect of interest of the object of study (for example, product reliability, process effectiveness), the perspective (for example, researcher or developer) and the context in which the measurement takes place [11].

Hypothesis “A hypothesis is a suggested explanation of a phenomenon or reasoned proposal suggesting a possible correlation between multiple phenomena” [166]. “A tentative explanation for an observation, phenomenon, or scientific problem that can be tested by further investigation [65].

Incremental development “A software development technique in which requirements definition, design, implementation and testing occur in an overlapping (rather than sequential) manner, resulting in incremental completion of the overall software product” [71].

Internal validity “...defines the degree of confidence in a cause-effect relationship between factors of interest and the observed results” [14].

Interpretivist “...believe all research must be interpreted within the context in which it takes place ...” [38]. Compare with *positivist*.

Interval scale “...defines a distance from one point to another, so that there are equal intervals between consecutive numbers. This property permits computations not available with the ordinal scale, such as calculating the mean. However, there is no absolute zero point in an interval scale, and thus ratios do not make sense. Care is thus needed when you make

comparisons.” “. . . we cannot say that today’s 30-degree Celsius temperature is twice as hot as yesterday’s 15 degrees” [132].

KiTe The name of the model presented in this thesis.

Language “A (formal) *language* is a set of strings from some one *alphabet*.” [62].

Measurement “a figure, extent or amount obtained by measuring” [70].

NASA/SEL “The Software Engineering Laboratory (SEL) is an organization sponsored by the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC) and created to investigate the effectiveness of software engineering technologies when applied to the development of applications software” [117].

NATO North Atlantic Treaty Organisation. “. . . an international organisation for defense collaboration established in 1949, in support of the *North Atlantic Treaty* . . .” [164].

Nominal scale “. . . puts items into categories, such as when we identify a programming language as Ada, Cobol, Fortran, or C++” [132].

Object In the context of this document, *object* denotes a person or thing in the real-world i.e. a material thing. The term is required to distinguish between abstract ideas e.g. state spaces and the ‘real’ objects which the spaces describe.

Ontology “An explicit formal specification of how to represent the objects, concepts and other entities that are assumed to exist in some area of interest and the relationships that hold among them.” “A set of agents that share the same ontology will be able to communicate about a domain of discourse without necessarily operating on a globally shared theory. We say that an agent commits to an ontology if its observable actions are consistent with the definitions in the ontology” [68].

Ordinal scale “. . . ranks items in an order, such as when we assign failures a progressive severity like minor, major and catastrophic” [132].

PMBOK Guide Guide to the Project Management Body of Knowledge. An “inclusive term that describes the sum of knowledge within the profession of project management” [135].

Positivist “. . . looks for irrefutable facts and fundamental laws that can be shown to be true regardless of the researcher and the occasion” [38]. Compare with *interpretivist*.

Prescriptive process A description of a *process* that takes into account only technical aspects and implicitly makes assumptions that human factors do not affect *process* outcomes. Compare with *realised process*.

Process “The sequence of activities, people, and systems involved in carrying out some business or achieving some desired result” [69]. “A series of actions, changes, or functions bringing about a result” [66].

Product The artifacts that implement a software-intensive system and are the deliverables from a *project*.

Product engineering “The technical processes to define, design, and construct or assemble a *product*“ [71].

Project “.. a temporary endeavour to create a unique service or product and with a definite beginning and end” [135].

Randomized experiment “An *experiment* in which units are assigned to receive the treatment or an alternative condition by a random process . . .” [149] quoting Shadish et. al.

Quasi-Experiment “An *experiment* in which units are not assigned to conditions randomly” [149] quoting Shadish et. al.

Ratio scale “. . . incorporates an absolute zero, preserves ratios, and permits the most sophisticated analysis. Measure such as lines of code or numbers of defects are ratio measures. It is for this scale that we can say that A is twice the size of B” [132].

Realised process A description of a *process* as it really happens i.e. that takes into account how all factors relevant to *process* outcomes, for example, the people involved and *project contexts*, affect these outcomes. Compare with *prescriptive process*.

Reliability “. . . the ability of a system or component to perform its required functions under stated conditions for a specified period of time” [71].

SCM Software Configuration Management [72].

Software design description (SDD) “A representation of a software system created to facilitate analysis, planning, implementation, and decision making. A blueprint or model of the software system. The SDD is used as the primary medium for communicating software design information” [73].

Software development “Any activity related to the production or modification of software pursuing some goal(s) beyond the software itself” [104]. This definition is broader than that given by IEEE [71], which states a systematic and quantifiable approach. The broader definition enables us to include more flexible and informal approaches.

Software development cycle “The period of time that begins with the decision to develop a software product and ends when the software is delivered“ [71].

Software development process The set of all activities that affect some representation of a software product.

Software life cycle “The period of time that begins when a software product is conceived and ends when the software is no longer available for use“ [71].

Software process “. . . the set of all activities which are carried out in the context of a concrete software development project. It usually covers aspects of software development, quality management, configuration management and project management” [59].

Task A piece of work carried out by one or more *engineers*.

Theory “. . . a theory is a proposed description, explanation, or model of the manner of interaction of a set of natural phenomena, capable of predicting future occurrences or observations of the same kind, and capable of being tested through experiment or otherwise falsified through empirical observation.” “A theory is a logically self-consistent model or framework for describing the behavior of a related set of natural or social phenomena.” [167].

Understandability “. . . the degree to which the purpose of the system or component is clear to the evaluator” [150].

Bibliography

- [1] T.K. Abdel-Hamid and S.E. Madnick. *Software Project Dynamics—An Integrated Approach*. Prentice-Hall, Inc., New Jersey, USA, 1991.
- [2] Zeiad Abdelnabi, Giovanni Cantone, Marcus Ciolkowski, and Dieter Rombach. Comparing Code Reading Techniques Applied to Object-Oriented Software Frameworks with regard to Effectiveness and Defect Detection Rate. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering (ISESE'04)*, pages 239–248. The Institute of Electrical and Electronic Engineers, Inc., 2004.
- [3] Silvia T. Acuna and Natalia Juristo. Modelling Human Competencies in the Software Process. In *Proceedings of the 2003 International Workshop on Software Process Simulation and Modeling (ProSim'03)*, Portland, Oregon, U.S.A., 2003.
- [4] Sylvia T. Acuna, Marta Gomez, and Natalia Juristo. Understanding Team Formation in Software Development. In *Proceedings of the 6th International Workshop on Software Process Simulation and Modeling (ProSim'05)*, pages 26–38, St. Louis, Missouri, 2005. Fraunhofer IRB.
- [5] Niniek Angkasaputra and Dietmar Pfahl. Towards an Agile Development Method of Software Process Simulation. In *Proceedings of the 6th International Workshop on Software Process Simulation and Modeling (ProSim'05)*, pages 83–94, St. Louis, Missouri, 2005. Fraunhofer IRB.
- [6] Phillip G. Armour. Not-Defect: The Mature Discipline of Testing. *Communications of the ACM*, 47(10):15–18, 2004.
- [7] Martin Auer and Stefan Biffel. Increasing the Accuracy and Reliability of Analogy-Based Cost Estimation with Extensive Project Feature Dimension Weighting. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering (ISESE'04)*. The Institute of Electrical and Electronic Engineers, Inc., 2004.

- [8] Jr. Augusto A. Legasto and James M. Lyneis. Introduction. In Augusto A. Legasto Jr., Jay W. Forrester, and James M. Lyneis, editors, *Studies in the Management Sciences Volume 14: System Dynamics*. North-Holland, The Netherlands, 1980.
- [9] Jr. Augusto A. Legasto and Joseph A. Maciariello. System Dynamics: A Critical Review. In Augusto A. Legasto Jr., Jay W. Forrester, and James M. Lyneis, editors, *Studies in the Management Sciences Volume 14: System Dynamics*. North-Holland, The Netherlands, 1980.
- [10] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. The Experience Factory. In John J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 1. John Wiley and Sons, Ltd, 1994.
- [11] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. The Goal Question Metric Approach. <http://www.wagse.informatik.uni-kl.de/pubs/repository/basili94b/encyclo.gqm.pdf>, 2006.
- [12] Victor R. Basili and Barry T. Perricone. Software Errors and Complexity: An Empirical Investigation. *Communications of the ACM*, 27(1):42–52, 1984.
- [13] Victor R. Basili and H. Dieter Rombach. Tailoring the Software Process to Project Goals and Environments. In *Proceedings of the Ninth International Conference on Software Engineering*. IEEE, IEEE Press, 1987.
- [14] Victor R. Basili, Forrest Shull, and Filippo Lanubile. Building Knowledge through Families of Experiments. *IEEE Transactions on Software Engineering*, 25(4), 1999.
- [15] Kent Beck. *eXtreme Programming eXplained - Embrace Change*. Addison-Wesley, United States of America, 2000.
- [16] Kent Beck and Barry Boehm. Agility through Discipline: A Debate. *IEEE Computer*, June 2003, 2003.
- [17] James A. Bell and Peter M. Senge. Methods for Enhancing Refutability in System Dynamics Modeling. In Augusto A. Legasto Jr., Jay W. Forrester, and James M. Lyneis, editors, *Studies in the Management Sciences Volume 14: System Dynamics*. North-Holland, The Netherlands, 1980.
- [18] Robert M. Bell, Thomas J. Ostrand, and Elaine J. Weyuker. Challenges in Predicting the Location of Faults in Large Software Systems. In *Proceedings of the 2004 Workshop on Predictive Software Models*. The Institute of Electrical and Electronic Engineers, Inc., 2004.

- [19] Barry Boehm. *Software Engineering Economics*. Prentice-Hall, Inc., 1981.
- [20] Barry Boehm. Value-Based Software Engineering. *ACM SIGSOFT Software Engineering Notes*, 28(2), 2003.
- [21] Barry Boehm and Richard Turner. *Balancing Agility and Discipline*. Addison-Wesley, 2003.
- [22] Barry Boehm and Richard Turner. Using Risk to Balance Agile and Plan-Driven Methods. *IEEE Computer*, June 2003, 36(6), 2003.
- [23] Barry W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, May(11), 1988.
- [24] Barry W. Boehm and Kevin J. Sullivan. Software economics: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 319–343, Limerick, Ireland, 2000. ACM Press.
- [25] Lionel C. Briand, Khaled El Emam, Dagmar Surmann, Isabella Wieczorek, and Katrina D. Maxwell. An Assessment and Comparison of Common Software Cost Estimation Modeling Techniques. In *Proceedings of the 1999 Conference on Software Engineering*. IEEE Computer Society Press, 1999.
- [26] Marvin J. Carr, Suresh L. Konda, Ira Monarch, F. Carol Ulrich, and Clay F. Walker. Taxonomy-Based Risk Identification. Technical Report CMU/SEI-93-TR-6, Software Engineering Institute, Carnegie Mellon University, 1993.
- [27] Jeff Carver, John Van Voorhis, and Victor Basili. Understanding the Impact of Assumptions on Experimental Validity. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering (ISESE'04)*. The Institute of Electrical and Electronic Engineers, Inc., 2004.
- [28] Zhihao Chen, Tim Menzies, Daniel Port, and Barry Boehm. Finding the Right Data for Software Cost Modeling. *IEEE Software*, November/December, 2005.
- [29] Alistair Cockburn. Agile Software Development The Cooperative Game. <http://alistair.cockburn.us/crystal/talks/sdacg/swdevasacooperativegame060.ppt>, 2005.
- [30] David Cohen, Mikael Lindvall, and Patricia Costa. Agile Software Development (Draft Version). Technical report, Fraunhofer Center for Experimental Software Engineering and The University of Maryland, 2003.

- [31] Theodore Coladarci, Casey D. Cobb, Edward W. Minium, and Robert C. Clarke. *Fundamentals of Statistical Reasoning in Education*. John Wiley and Sons, Ltd, USA, 2004.
- [32] Ward Cunningham. Manifesto for agile software development. <http://agilemanifesto.org/>, 2001.
- [33] B. Curtis and D. Walz. The Psychology of Programming in the Large: Team and Organizational Behaviour. In J.-M. Hoc, T.R.G. Green, R. Samurcay, and D.J. Gilmore, editors, *Psychology of Programming*, pages 253–270. Academic Press Ltd., London, U.K., 1990.
- [34] Bill Curtis. Three Problems Overcome with Behavioral Models of the Software Development Process. In *Proceedings of the Eleventh International Conference on Software Engineering*, pages 398–399, 1989.
- [35] Bill Curtis, Marc I. Kellner, and Jim Over. Process Modeling. *Communications of the ACM*, 35(9):75–90, 1992.
- [36] Bill Curtis, Herb Krasner, and Neil Iscoe. A Field Study of the Software Design Process for Large Systems. *Communications of the ACM*, 31(11):1268–1287, 1988.
- [37] Bill Curtis, Herb Krasner, Vincent Shen, and Neil Iscoe. On Building Software Process Models Under the Lamppost. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 96–105. IEEE, IEEE Computer Society Press, 1987.
- [38] Ray Dawson, Phil Bones, Briony J. Oates, Pearl Brereton, Motoei Azuma, and Mary Lou Jackson. Empirical Methodologies in Software Engineering. In *Proceedings of the Eleventh Annual International Workshop on Software Technology and Engineering Practice*. The Institute of Electrical and Electronic Engineers, Inc., 2004.
- [39] Marcio de Oliveira Barros, Claudia Maria Lima Werner, and Guilherme Horta Travassos. System Dynamics Extension Modules for Software Process Modeling. In *Proceedings of the 2003 International Workshop on Software Process Simulation and Modeling (ProSim'03)*, Portland, Oregon, U.S.A., 2003.
- [40] M. Deck and B.E. Hines. Cleanroom software engineering for flight systems: A preliminary report. In *Proc. of 1997 IEEE Aerospace Conference*, volume 4, pages 329–347. IEEE Computer Society Press, 1997.
- [41] Michael D. Deck. Cleanroom Software Engineering: Quality Improvement and Cost Reduction. In *Proc. Pacific Northwest Software Quality Conference*, pages 243–258, 1994.

- [42] Peter DeGrace and Leslie Hulet Stahl. *Wicked Problems, Righteous Solutions: A Catalogue of Modern Software Engineering Paradigms*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1991.
- [43] Yogesh Deshpande, San Murugesan, and Steve Hansen. Web Engineering: Beyond CS, IS and SE Evolutionary and Non-engineering Perspectives. In San Murugesan and Yogesh Deshpande, editors, *Web Engineering*, pages 14–23. Springer-Verlag, Germany, 2001.
- [44] Beverly R. Dixon and Gary D. Bouma. *The Research Process*. Oxford University Press, Melbourne, 1984.
- [45] Paolo Donzelli and Guiseppe Iazeolla. Hybrid simulation modelling of the software process. *Journal of Systems and Software*, 59(3/3), 2001.
- [46] A. Drappa and J. Ludewig. Quantitative modeling for the interaction simulation of software projects. *Journal of Systems and Software*, 46(2/3), 1999.
- [47] Mike Falla. Advances in Safety Critical Systems. Technical report, Department of Trade and Industry (DTI) and the Engineering and Physical Sciences Research Council (EPSRC), 1997. Results and Achievements from the DTI/EPSC RandD Programme in Safety Critical Systems.
- [48] Norman Fenton and Martin Neil. New Directions in Software Metrics. http://www.dcs.qmw.ac.uk/~norman/papers/new_directions_metrics/start.htm, 1999.
- [49] Norman Fenton, Shari Lawrence Pfleeger, and Robert Glass. Science and Substance: A Challenge to Engineers. *IEEE Software*, July, 1994.
- [50] Norman E. Fenton and Niclas Ohlsson. Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Transactions on Software Engineering*, 26(8), 2000.
- [51] Susan Ferreira and James Collofello, et al. Utilization of Process Modeling and Simulation in Understanding the Effects of Requirements Volatility in Software Development. In *Proceedings of the 2003 International Workshop on Software Process Simulation and Modeling (ProSim'03)*, Portland, Oregon, U.S.A., 2003.
- [52] Jay W. Forrester. *Principles of Systems*. Productivity Press, Portland, Oregon, USA, 1971.

- [53] Martin Fowler. The New Methodology. <http://www.martinfowler.com/articles/newMethodology.html>, 2005.
- [54] Alfonso Fuggetta. Rethinking the modes of software engineering research. *Journal of Systems and Software*, 46(2/3), 1999.
- [55] David J. Gilmore. Methodological Issues in the Study of Programming. In J.-M. Hoc, T.R.G. Green, R. Samurcay, and D.J. Gilmore, editors, *Psychology of Programming*, pages 83–98. Academic Press Ltd., London, U.K., 1990.
- [56] David P. Gluch. A Construct for Describing Software Development Risks. Technical Report CMU/SEI-94-TR-14, Software Engineering Institute, Carnegie Mellon University, 1994.
- [57] Ian Graham, Brian Henderson-Sellars, and Houman Younessi. *OPEN Process Specification*. Addison-Wesley, 1997.
- [58] Todd L. Graves, Alan F. Karr, J.S. Marron, and Harvey Sly. Predicting Fault Incidence Using Software Change History. *IEEE Transactions on Software Engineering*, 26(7), 2000.
- [59] Volker Gruhn. Process-Centred Software Engineering Environments A Brief History and Future Challenges. *Annals of Software Engineering*, 14, 2002.
- [60] Len Hatton. Reexamining the Fault Density-Component Size Connection. *IEEE Software*, March/April, 1997.
- [61] Jim Highsmith. History: The Agile Manifesto. <http://agilemanifesto.org/history.html>, 2001.
- [62] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, United States of America, 1979.
- [63] Houghton Mifflin Company. The American Heritage Dictionary of the English Language Fourth Edition - Engineering. <http://dictionary.reference.com/search?q=engineering>, 2000.
- [64] Houghton Mifflin Company. The American Heritage Dictionary of the English Language Fourth Edition - Evidence. <http://dictionary.reference.com/search?q=evidence>, 2000.
- [65] Houghton Mifflin Company. The American Heritage Dictionary of the English Language Fourth Edition - Hypothesis. <http://dictionary.reference.com/search?q=hypothesis>, 2000.

- [66] Houghton Mifflin Company. The American Heritage Dictionary of the English Language Fourth Edition - Process. <http://dictionary.reference.com/search?q=process>, 2000.
- [67] Watts S. Humphrey and Marc I. Kellner. Software Process Modeling: Principles of Entity Process Models. Technical report, Software Engineering Institute, 1989. Report appears in the Proceedings of the 11th International Conference on Software Engineering, May 1989 (ICSE 11).
- [68] Imperial College London, Department of Computing. Free On-Line Dictionary of Computing. <http://foldoc.doc.ic.ac.uk/foldoc/foldoc.cgi?query=ontology&action=Search>, 1997.
- [69] Imperial College London, Department of Computing. Free On-Line Dictionary of Computing. <http://dictionary.reference.com/search?q=process>, 2005.
- [70] Imperial College London, Department of Computing. Free On-Line Dictionary of Computing. <http://dictionary.reference.com/search?q=measurement>, 2005.
- [71] Institute of Electrical and Electronic Engineers. Std 610.12-1990: IEEE Standard Glossary of Software Engineering Terminology. In *IEEE Standards Collection - Software Engineering*. The Institute of Electrical and Electronic Engineers, Inc., New York, USA, 1990.
- [72] Institute of Electrical and Electronic Engineers. Std 828-1990: IEEE Standard for Software Configuration Management Plans. In *IEEE Standards Collection - Software Engineering*. The Institute of Electrical and Electronic Engineers, Inc., New York, USA, 1990.
- [73] Institute of Electrical and Electronic Engineers. Std 1016-1998: IEEE Recommended Practice for Software Design Descriptions. In *IEEE Standards Collection - Software Engineering*. The Institute of Electrical and Electronic Engineers, Inc., New York, USA, 1999.
- [74] Institute of Electrical and Electronic Engineers. Std 830-1998: IEEE Recommended Practice for Software Requirements Specifications. In *IEEE Standards Collection - Software Engineering*. The Institute of Electrical and Electronic Engineers, Inc., New York, USA, 1999.

- [75] Integrated Computer Engineering, Inc. Software Program Managers Network. <http://www.spmn.com>, 2005.
- [76] J.-M. Hoc and T.R.G. Green and R. Samurcay and D.J. Gilmore. Theoretical and Methodological Issues. In J.-M. Hoc, T.R.G. Green, R. Samurcay, and D.J. Gilmore, editors, *Psychology of Programming*, pages 3–7. Academic Press Ltd., London, U.K., 1990.
- [77] Randall W. Jensen. Extreme Software Cost Estimating. *CrossTalk: The Journal of Defense Software Engineering*, January, 2004.
- [78] Bary Alyssa Johnson. Students Design Future Phone Concepts. *PC Magazine*, 25(3), 2006.
- [79] Capers Jones. *Programming Productivity*. McGraw-Hill, Inc, 1986.
- [80] Praveen Kallakuri and Sebastian Elbaum. Experimental Studies in Empirical Software Engineering. <http://www.acm.org/crossroads/xrds7-4/empirical.html>, 2003.
- [81] Marc I. Kellner, Raymond J. Madachy, and David M. Raffo. Software Process Simulation Modelling: Why? What? How? *Journal of Systems and Software*, 46(2/3), 1999.
- [82] Chris F. Kemerer. An Empirical Validation of Software Cost Estimation Models. *Communications of the ACM*, 30(5):416–429, 1987.
- [83] Taghi M. Khoshgoftaar, Edward B. Allen, Kalai S. Kalaichelvan, and Nishith Goel. Early Quality Prediction: A Case Study in Telecommunications. *IEEE Software*, January, 1996.
- [84] Diana Kirk and Ewan Tempero. Supporting Empirical Software Process Research through Process Modelling. Technical Report UoA-SE-2005-8, University of Auckland, 2005.
- [85] Diana Kirk and Ewan Tempero. Identifying Risks in XP Projects through Process Modelling. In *Proceedings of the Australian Software Engineering Conference (ASWEC'06)*, pages 411–420, Sydney, Australia, 2006. IEEE Computer Society.
- [86] Barbara Kitchenham. The Problem with Function Points. *IEEE Software*, 14(2), 1997.
- [87] Barbara Kitchenham, David Budgen, Pearl Brereton, and Stephen Linkman. Realising Evidence-Based Software Engineering. In *Realising Evidence-Based Software Engineering Workshop 2005, Workshop co-located with ICSE 2005*, St. Louis, Missouri, 2005. Keele University.

- [88] Barbara Kitchenham and Roland Cairn. Research and Practice: Software Design methods and Tools. In J.-M. Hoc, T.R.G. Green, R. Samurcay, and D.J. Gilmore, editors, *Psychology of Programming*, pages 271–284. Academic Press Ltd., London, U.K., 1990.
- [89] Barbara Kitchenham, Shari Lawrence Pfleeger, and Norman Fenton. Towards a Framework for Software Measurement Validation. *IEEE Transactions on Software Engineering*, 21(12), 1995.
- [90] Barbara A. Kitchenham, Shari Lawrence Pfleeger, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary Guidelines for Empirical Research in Software Engineering. *IEEE Transactions on Software Engineering*, 28(8), 2002.
- [91] Erhard Konrad. Software Measurement Misguided. <http://www.wbs.cs.tu-berlin.de/user-taipan/erhard/contributions.html>, 2006.
- [92] Philippe Kruchten. *The Rational Unified Process: An Introduction, Second Edition*. Addison-Wesley, United States of America, 2000.
- [93] Peter B. Lakey. A Hybrid Software Process Simulation Model for Project Management. In *Proceedings of the 2003 International Workshop on Software Process Simulation and Modeling (ProSim'03)*, Portland, Oregon, U.S.A., 2003.
- [94] Filippo Lanubile and Giuseppe Visaggio. Evaluating Predictive Quality Models Derived from Software Measures: Lessons Learned. *Journal of Systems and Software*, 38, 1997.
- [95] Craig Larman and Victor R. Basili. Iterative and Incremental Development: A Brief History. *IEEE Computer*, June 2003, 2003.
- [96] Meir M. Lehman, Dewayne E. Perry, and Wladyslaw M. Turski. Why is it so hard to find Feedback Control in Software Processes? In *Proceedings of the 19th Australasian Computer Science Conference, Melbourne, Australia*, 1996.
- [97] M.M. Lehman. Process Models, Process Programs, Programming Support. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 14–16. IEEE, IEEE Computer Society Press, 1987.
- [98] M.M. Lehman. Process Modelling - Where Next. In *Proceedings of the 1997 Conference on Software Engineering*. IEEE Computer Society Press, 1997.
- [99] M.M. Lehman and J. F. Ramil. The impact of feedback in the global software process. *Journal of Systems and Software*, 46(2/3), 1999.

- [100] Richard C. Linger. Cleanroom Process Model. *IEEE Software*, March, 1994.
- [101] Martin Lippert, Petra Becker-Pechau, Holger Breitling, Koch Koch, Andreas Kornstadt, Stefan Roock, Axel Schmolitzky, Henning Wolf, and Heinz Zullighoven. Developing Complex Projects Using XP with Extensions. *IEEE Computer*, 36(6), 2003.
- [102] Todd Little. Value Creation and Capture: A Model of the Software Development Process. *IEEE Software*, May/June, 2004.
- [103] Bernard Londeix. *Cost Estimation for Software Development*. Addison-Wesley, Cornwall, UK, 1987.
- [104] Jochen Ludewig. Models in Software Engineering - an introduction. *Software and Systems Modeling*, 2(1), 2003.
- [105] R.J. Madachy. System dynamics modelling of an inspection-based process. In *Proc. Eighteenth ICSE*, pages 376–386, Berlin, Germany, 1996. IEEE Computer Society Press.
- [106] Robert Martin and David Raffo. A Model of the Software Development Process Using Both Continuous and Discrete Models. *International Journal of Software Process Improvement and Practice*, 5(2/3), 2000.
- [107] Robert H. Martin and David M. Raffo. Application of a hybrid process simulation model to a software development project. *Journal of Systems and Software*, 59(3/3), 2001.
- [108] Pete McBreen. *Software Craftmanship: The New Imperative*. Addison-Wesley, USA, 2002.
- [109] Marco Melis, Ivana Turnu, Alessandra Cau, and Giulio Concas. A Software Process Simulation Model of Extreme Programming. In *Proceedings of the 6th International Workshop on Software Process Simulation and Modeling (ProSim'05)*, pages 63–70, St. Louis, Missouri, 2005. Fraunhofer IRB.
- [110] Manoel G. Mendonca and Victor R. Basili. Validation of an Approach for Improving Existing Measurement Frameworks. *IEEE Transactions on Software Engineering*, 26(6), 2000.
- [111] Peiwei Mi and Walt Scacchi. A Knowledge-Based Environment for Modeling and Simulating Software Engineering Processes. *IEEE Transaction on Knowledge and Data Engineering*, 2(3), 1990.
- [112] K.H. Moller and D. Paulish. An Empirical Investigation of Software Fault Distribution. *Software Quality Assurance and Measurement*, 1995.

- [113] Sandro Morasca, Lionel C. Briand, Victor R. Basili, Elaine J. Weyuker, and Marvin V. Zelkowitz. Comments on "Towards a Framework for Software Measurement Validation". *IEEE Transactions on Software Engineering*, 23(3), 1997.
- [114] Jurgen Munch. Transformation-based Creation of Custom-tailored Software Process Models. In *Proceedings of the 5th International Workshop on Software Process Simulation and Modeling (ProSim'04)*, Edinburgh, Scotland, 2004.
- [115] Jurgen Munch. Goal-oriented Composition of Software Process Patterns. In *Proceedings of the 6th International Workshop on Software Process Simulation and Modeling (ProSim'05)*, pages 164–168, St. Louis, Missouri, 2005. Fraunhofer IRB.
- [116] John C. Munson and Taghi M. Khoshgoftaar. The Detection of Fault-Prone Programs. *IEEE Transactions on Software Engineering*, 18(5), 1992.
- [117] National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC). <http://www.nasa.gov/centers/goddard/home/index.html>, 2006.
- [118] Peter Naur and Brian Randell. NATO Software Engineering Conference 1968. Conference report, NATO Science Committee, 1969. Report on a conference sponsored by the NATO SCIENCE COMMITTEE held in Garmisch, Germany, in October 1968.
- [119] Holger Neu, Thomas Hanne, Jurgen Munch, Stefan Nickel, and Andreas Wirsén. Creating a Code Inspection Model for Simulation-based Decision Support. In *Proceedings of the 2003 International Workshop on Software Process Simulation and Modeling (ProSim'03)*, Portland, Oregon, U.S.A., 2003.
- [120] John T. Nosek. The Case for Collaborative Programming. *Communications of the ACM*, 41(3), 1998.
- [121] Ken Orr. Agile Requirements: Opportunity or Oxymoron? *IEEE Software*, May/June, 2004.
- [122] Leon Osterweil. Software Processes are Software Too. In *Proceedings of the Ninth International Conference on Software Engineering*. IEEE, IEEE Press, 1987.
- [123] Thomas J. Ostrand and Elaine J. Weyuker. The Distribution of Faults in a Large Industrial Software System. In George S. Avrunin and Gregg Rothermel, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis*, pages 56–64. ACM, 2002.

- [124] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Where the Bugs Are. In George S. Avrunin and Gregg Rothermel, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2004.
- [125] David Lorge Parnas. The Limits of Empirical Studies of Software Engineering. In *Proceedings of the 2003 International Symposium on Empirical Software Engineering (ISESE'03)*, pages 2–5. The Institute of Electrical and Electronic Engineers, Inc., 2003.
- [126] Allen Parrish, Randy Smith, David Hale, and Joanne Hale. A Field Study of Developer Pairs: Productivity Impacts and Implications. *IEEE Software*, September/October, 2004.
- [127] Ricardo Peculis. Who Dares, Wins: A Chaos Management Approach for Software Development. <http://www.matra.com.au/~rpeculis/wdw.htm>, 2003.
- [128] Nancy Pennington and Beatrice Grabowski. The Tasks of Programming. In J.-M. Hoc, T.R.G. Green, R. Samurcay, and D.J. Gilmore, editors, *Psychology of Programming*, pages 45–62. Academic Press Ltd., London, U.K., 1990.
- [129] D. Pfahl and K. Lebsanft. Using simulation to analyse the impact of software requirement volatility on project performance. *Information and Software Technology*, 42, 2000.
- [130] Dietmar Pfahl and Karl Lebsanft. Integration of System Dynamics Modelling with Descriptive Process Modelling and Goal-oriented Measurement. *Journal of Systems and Software*, 46(2/3), 1999.
- [131] Shari Lawrence Pfleeger. Soup or Art? The Role of Evidential Force in Empirical Software Engineering. *IEEE Software*, January/February, 2005.
- [132] Shari Lawrence Pfleeger, Ross Jeffery, Bill Curtis, and Barbara Kitchenham. Status Report on Software Measurement. *IEEE Software*, 14(2), 1997.
- [133] Lesley M. Pickard, Barbara A. Kitchenham, and Peter W. Jones. Combining empirical results in software engineering. *Information and Software Technology*, 40, 1998.
- [134] Antony Powell, Keih Mander, and Duncan Brown. Strategies for lifecycle concurrency and iteration - A system dynamics approach. *Journal of Systems and Software*, 46, 1999.
- [135] Project Management Institute. *A Guide to the Project Management Body of Knowledge (PMBOK Guide)*. Project Management Institute, Inc., USA, 2000.
- [136] David Raffo and Warren Harrison. Combining Process Feedback with Discrete Event Simulation Models to Support Software Project Management. In *FEAST 2000*, London, 2000.

- [137] David M. Raffo, Joseph V. Vandeville, and Robert H. Martin. Software process simulation to achieve higher CMM levels. *Journal of Systems and Software*, 46(2/3), 1999.
- [138] Juan F Ramil and Neil Smith. Qualitative Simulation and the Study of Software Evolution. In *Proceedings of the 2003 International Workshop on Software Process Simulation and Modeling (ProSim'03)*, Portland, Oregon, U.S.A., 2003.
- [139] Patrick Rivett. *Principles of Model Building: The Construction of Models for Decision Analysis*. John Wiley and Sons, Ltd, Bath, Great Britain, 1972.
- [140] Friedrich Rosenkranz. *An Introduction to Corporate Modeling*. Duke University Press, Durham, North Carolina, 1979.
- [141] Winston Royce. Managing the Development of Large Software Systems. In *Proceedings, IEEE WestCon*, pages 328–339. The Institute of Electrical and Electronic Engineers, Inc., 1970.
- [142] Philip Sallis, Graham Tate, and Stephen MacDonell. *Software Engineering: Practice, Management, Improvement*. Addison-Wesley, Sydney, Australia, 1995.
- [143] Walt Scacchi. Experience with software process simulation and modelling. *Journal of Systems and Software*, 46(2/3), 1999.
- [144] Walt Scacchi. Process Models in Software Engineering. In J. Marciniak, editor, *Encyclopedia of Software Engineering (2nd. edn.)*. John Wiley and Sons, Ltd, 2001.
- [145] Thomas J. Schriber and Daniel T. Brunner. Inside Discrete-Event Simulation Software: How It Works And Why It Matters. In J.A.Jones, R.R.Barton, K.Kang, and P.A.Fishwick, editors, *Proceedings of the 2000 Winter Simulation Conference*, 2000.
- [146] Ed Seidewitz. What Models Mean. *IEEE Software*, May/June, 2003.
- [147] Richard W. Selby, Victor R. Basili, and F. Terry Baker. Cleanroom Software Development: An Empirical Evaluation. *Transactions on Software Engineering*, SE-13(9), 1987.
- [148] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., 1996.
- [149] Dag I.K. Sjoberg, Jo E. Hannay, Ove Hansen, Vigdis By Kampenes, Amela Karahasanovic, and Nils-Kristian Liborg amd Anette C. Rekdal. A Survey of Controlled Experiments in Software Engineering. *IEEE Transactions on Software Engineering*, 31(9), 2005.

- [150] Software Engineering Institute. Cleanroom Software Engineering. <http://www.sei.cmu.edu/str/descriptions/cleanroombody.html>, 2005.
- [151] Patrick J. Starr. Modeling Issues and Decisions in System Dynamics Modeling. In Augusto A. Legasto Jr., Jay W. Forrester, and James M. Lyneis, editors, *Studies in the Management Sciences Volume 14: System Dynamics*. North-Holland, The Netherlands, 1980.
- [152] Douglas M. Stewart. Operations Psychology. *Decision Line*, 28(5), 1987.
- [153] Harald Storrie. Making Agile Processes Scalable. In *Proceedings of the 2003 International Workshop on Software Process Simulation and Modeling (ProSim'03)*, Portland, Oregon, U.S.A., 2003.
- [154] System Dynamics Society. What is System Dynamics. <http://www.systemdynamics.org/>, 2006.
- [155] Kasey Thompson. People Projects: Psychometric Profiling. *Crosstalk - The Journal of Defense Software Engineering*, April, 2003.
- [156] Hans van Vliet. *Software Engineering: Principles and Practice*. John Wiley and Sons, Ltd, Chichester, England, 2000.
- [157] June M. Verner and William M. Evenco. In-House Software Development: What Project Management Practices Lead to Success? *IEEE Software*, January/February, 2005.
- [158] Wayne Wakeland, Robert H. Martin, and David Raffo. Using Design of Experiments, Sensitivity Analysis, and Hybrid Simulation to Evaluate Changes to a Software Development Process: A Case Study. In *Proceedings of the 2003 International Workshop on Software Process Simulation and Modeling (ProSim'03)*, Portland, Oregon, U.S.A., 2003.
- [159] W.D. Wallis. *A Beginner's Guide to Discrete Mathematics*. Birkhauser, Boston, USA, 2003.
- [160] Rob Weaver, Georgios Despotou, Tim Kelly, and John McDermid. Combining Software Evidence - Arguments and Assurance. In *Realising Evidence-Based Software Engineering Workshop 2005, Workshop co-located with ICSE 2005*, St. Louis, Missouri, 2005. Keele University.
- [161] Gerald M. Weinberg. *The Psychology of Computer Programming*. van Nostrand Reinhold, New York, USA, 1971.

- [162] P. Wernick and M.M. Lehman. Software process white box modelling for FEAST/1. *Journal of Systems and Software*, 46(2/3), 1999.
- [163] Wikimedia Foundation. Engineering. <http://en.wikipedia.org/wiki/Engineering>, 2005.
- [164] Wikimedia Foundation. Nato. <http://en.wikipedia.org/wiki/NATO>, 2005.
- [165] Wikimedia Foundation. Causality. <http://en.wikipedia.org/wiki/Causality>, 2006.
- [166] Wikimedia Foundation. Hypothesis. <http://en.wikipedia.org/wiki/Hypothesis>, 2006.
- [167] Wikimedia Foundation. Theory. <http://en.wikipedia.org/wiki/Theory>, 2006.
- [168] Laurie Williams and Alistair Cockburn. Agile Software Development: It's about Feedback and Change. *IEEE Computer*, June 2003, 2003.
- [169] Laurie Williams, Robert R. Kessler, Ward Cunningham, and Ron Jeffries. Strengthening the Case for Pair Programming. *IEEE Software*, July/August, 2000.
- [170] Laurie Williams, Lucas Layman, and Pekka Abrahamsson. On Establishing the Essential Components of a Technology-dependent Framework: A Strawman Framework for Industrial Case Study-Based Research. In *Realising Evidence-Based Software Engineering Workshop 2005, Workshop co-located with ICSE 2005*, St. Louis, Missouri, 2005. Keele University.
- [171] Jim Woodcock and Jim Davies. *Using Z - Specification, Refinement and Proof*. Prentice-Hall, Inc., 1996.
- [172] Tze-Jie Yu, Vincent Y. Shen, and Hubert E. Dunsmore. An Analysis of Several Software Defect Models. *Transactions on Software Engineering*, 14(9), 1988.
- [173] Xuemei Zhang and Hoang Pham. The analysis of factors affecting software reliability. *Journal of Systems and Software*, 50, 2000.